Fachhochschule Dortmund
University of Applied Sciences and Arts

MASTER THESIS

Power Simulation of a MIPS microAptiv UP Core implemented as a virtual ASIC prototype in a 65nm CMOS technology

Author:
Yanchen Shi

Supervisor:
Prof. Dr. -Ing. Michael Karagounis

Advisor:
M. Eng. Alexander Walsemann

April 14, 2022

Abstract

This thesis presents a power simulation of a MIPS MicroAptiv UP Core implemented as a virtual ASIC prototype using Taiwan Semiconductor Manufacturing Company(TSMC) 65 nm CMOS technology. Based on the MIPS instruction set program data is generated and introduced in the simulation by means of initialization files. Before the simulation, technology specific SRAM modules are integrated into the MIPS core. Two different programs are used for power characterization. The first program performs frequent memory accesses by means of load/store word instructions, while the second program is a loop which operates on registers only and mainly increments addresses. The simulation is based on a virtual prototype which is generated by synthesis and place & route including post-layout parasitic extractions. The stimuli for the power extraction is generated via gate-level simulation and forwarded to the power calculation engine. The effect of X-propagation on gate-level simulations is avoided by modifying the address-related statements in the execution data path module, which use another form of 2 to 1 multiplexer, setting the output to zero for all input signals even with an initial value of 'x' without changing the functionality. Finally, the consumed power is provided by reports generated by the power simulation engine. The memory-centric program consumes 35.39mW of internal power using instructions, which is 0.73mW less than the internal power of the register-centric program, and the overall average power is also lower by almost 0.7mW.

Keywords—Power Simulation; TSMC 65nm CMOS; X-propagation; MIPS

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Listings

1 INTRODUCTION

The MIPS32® microAptiv™ UP core is a high-performance, low-power, 32-bit MIPS RISC processor core intended for custom system-on-silicon applications. The core is designed for semiconductor manufacturing companies, ASIC developers, and system OEMs who want to rapidly integrate their custom logic and peripherals with a high-performance RISC processor. The microAptiv UP core is fully synthesizable to allow maximum flexibility. It is highly portable across processes and can easily be integrated into full system-on-silicon designs. The microAptiv UP core implements the MIPS Architecture in a 5-stage pipeline. It includes support for the micro-MIPS™ ISA, an Instruction Set Architecture with optimized MIPS32 16-bit and 32-bit instructions that provide a significant reduction in code size with a performance equivalent to MIPS32.
In this thesis, the primary modules, their basic function, register set, and instruction set are introduced in the second chapter. The reason for describing these modules in this text is to help the reader to understand the architecture and operation mode of the MIPS core. In addition the given information is also applied in exemplary assembly programs. Some other modules not mentioned in this text can be found in [1]. As an additional note, the figures in this chapter are captured from the Vivado software, while in chapters 5 and 6 Cadence software is used. In the third chapter, three different algorithms for the virtual-to-physical translation are presented. This includes the "translation of the unmapped segment," LTB, and FMT approach. Chapter 4 shows the exact process for the translation of the unmapped segment. In addition, the five relevant stages for the execution of the *lw* instruction are also explained. The two different processes for the translation between data virtual address and instruction virtual address, and the process of read/write data are described in this chapter which is also useful for finding undefined signals during gate-level-simulation in chapter 6. The next chapter focuses on replacing generic memory modules with technology specific memories from TSMC and how to generate the program files to be loaded into the memory cells during simulation. The RTL or functional simulation results before synthesis, floorplan, and P&R process are briefly described. The instructions for defining timing constraints critical to the clock tree and the content of the constraints file are also presented in the synthesis section. The last chapter introduces the basic setup of the gate-level-simulation. It also describes problems and results encountered during this simulation. Finally a comparison and analysis of the results for the two cases of power simulation is given.

2 ARCHITECTURE OF THE MIPS

This chapter introduces the core, the register set, the instruction set, and the special microarchitecture of the MIPS processor. The remainder of this chapter introduces all components of the MIPS model as well as the function of these components.

2.1 Block diagram of the core

The MIPSfpga core is freely available and has a MIPS32 microAptiv UP architecture. The main parts of the MIPSfpga core are shown in the following block diagram illustrated in Figure 2.1. The microAptiv UP core is designed with a 5-stage instruction pipeline. It includes support

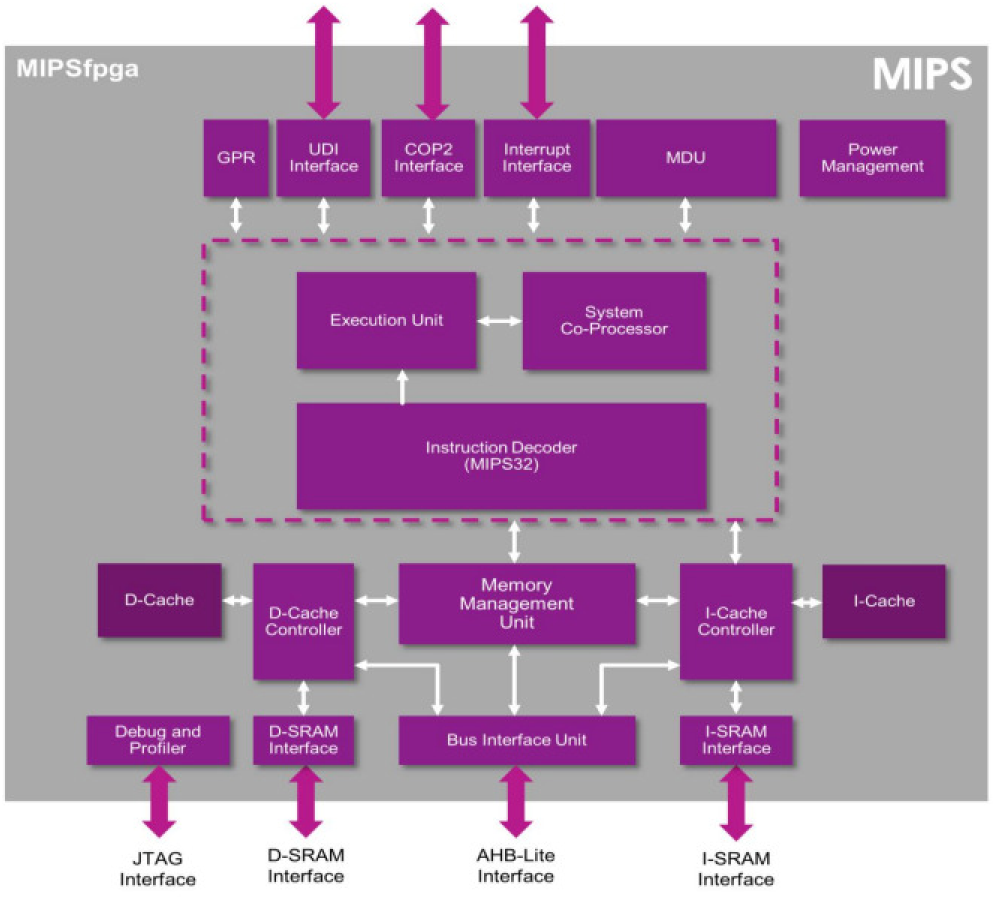


Figure 2.1: Block diagram of MIPS32 microAptiv UP core [2]

for the microMIPS ISA, which is an instruction set architecture with optimized 16-bit and 32-bit instructions. The Instruction Decoder gets the actual instruction from the instruction cache, which acts as a buffer memory between external memory and the core processor, and is an additional hierarchy level of memory to access instructions faster. The Instruction Decoder generates signals that interact with the Execution Unit to trigger the execution of the decoded operation. The Execution Unit has a load/store architecture and is equipped with a single-cycle ALU to perform logical, shift, and mathematical operations. In addition, it has an autonomous multiply/divide unit. It also includes an Address Unit, which determines the next Program Counter (PC) value by controlling address selections muxes and reacting on the branch condition as well as the trap condition comparator. The 32-bit General Purpose Registers (GPR) are used for integer operations and address calculations. To minimize the context switching overhead during interrupt or exception processing one, three, seven or fifteen additional shadow registers files can be added. The System Coprocessor unit provides system interface signals, such as the system clock and reset. The MMU (Memory Management Unit), which is connected to the instruction-, and data cache controller (I-Cache and D-Cache) and also the Bus Interface Unit performs virtual-to-physical address translation and takes the instructions or data from the main memory when these data is not available in the caches. The Bus Interface Unit to which the MMU is connected to allows the user to access memories and memory-mapped I/O through an AHBLite bus. The Multiply & Divide Unit (MDU) performs multiply/divide operations.

The MIPS architecture offers up to four different coprocessors (CP0-CP3). However, only the CP0 coprocessor is mandatory, while the others are optional. In this thesis, only the CP0 (system coprocessor) is used, which translates virtual addresses into physical addresses, manages exceptions, and handles switches between kernel supervisor and users states. Furthermore, the CP0 controls the cache subsystem and provides diagnostic control and error recovery facilities. The CP1-CP3 coprocessor slots are usually reserved for the floating-point calculations.

2.2 PIPELINE MICROARCHITECTURE

Pipelining is used to improve sequential logic by splitting combinational logic blocks into several smaller segments. In between the segments, registers are placed which store the segment outputs with every clock cycle and feed the inputs of the subsequent segment. By this means the stability and the consistency of the logic operation can be secured and efficiency and the processing speed can be increased. A five-stage instruction pipeline is applied in the case of the MIPS microprocessor core. As shown in Table 2.1, the five stages are Fetch, Decode, Execute, Memory and Writeback. The processor reads the instructions from the instruction memory during the Fetch stage. During the Decode phase, the fetched instruction is decoded to define the sources of the operands in the register file and to produce control signals for the subsequent execution of the decoded instruction. While in the Execution phase, the processor performs a computation using the resources available in its ALU. Data memory can be read or written by the processor in the Memory stage. The operation result can be written by the processor to the register file in the Writeback stage.

IF	ID	EXE	MEM	WB		
	IF	ID	EXE	MEM	WB	
		IF	ID	EXE	MEM	WB

Table 2.1: Pipeline processor

In each stage the next instruction is taken over as soon as the processing of the current instruction is finished. Registers are required between each pipeline stage to hold the result of the executed instruction. The results stored in these registers are used for processing the next pipeline stage.

2.3 MEMORY MAP

The microAptiv UP core provides a 32-bit address space and three modes of operation, namely User mode, Kernel mode, and Debug mode. The processing mode influences translation of virtual addresses to physical addresses, which the MMU performs. The translation process happens before a request is sent to the cache controllers and the bus interface unit for external memory access.

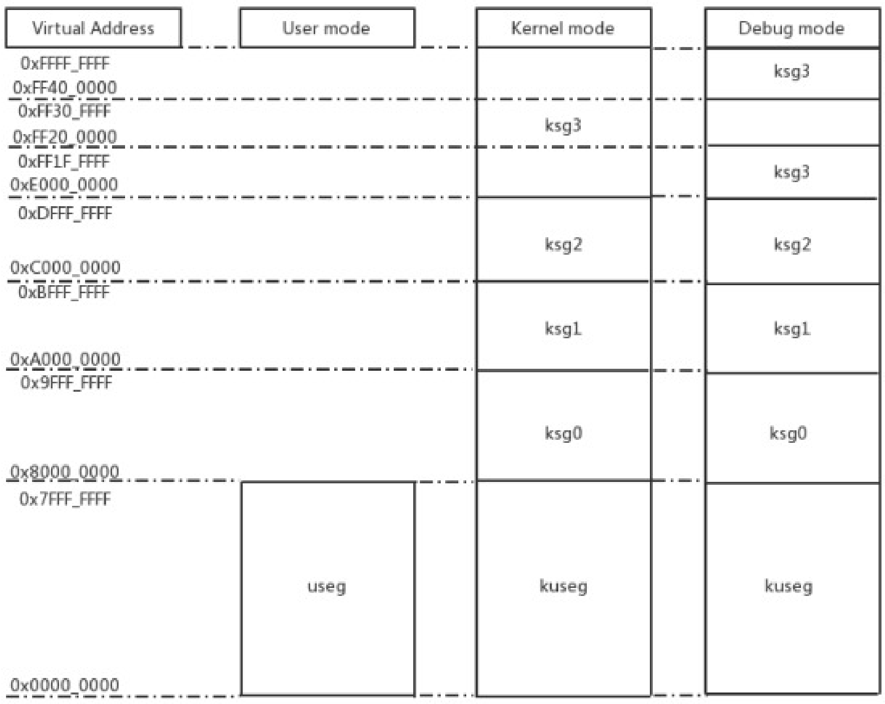


Figure 2.2: Memory map [1]

- User mode is applied during the execution of application programs.
- Kernel-mode is used for handling exceptions and operating system kernel functions, including CP0 management and I/O device access.
- Debug mode is used during system bring-up and software development.

When a reset or another exception is accepted, the core uses Kernel mode, in which the software can operate in the entire address space, and also the CP0 register. In User mode, only part of the virtual address space from *0x0000_0000* to *0x7FFFF_FFFF* is available. In the opposite to Kernel-mode, the CP0 functions can not be accessed. The remaining address space from *0x8000_0000* to *0xFFFF_FFFF* is only accessible to exceptions.

Debug mode is entered by triggering the debug exception. In Debug mode, the same address space and the CP0 registers are available as in Kernel mode. In segment kseg3, the core has an additional segment dseg, which can be turned on or off.

2.4 AHB-Lite Bus

As shown in Figure 2.3, the available AHB-Lite interface consists of a 50MHz system clock *HCLK*, a write enable signal *HWRITE* ("1"write,"0"read), an address signal bus *HADDR [31:0]*, and two separate read and write data buses *HRDATA[31:0]* and *HWDATA[31:0]*. The memories and all peripherals are connected to the core through the AHB-Lite interface.

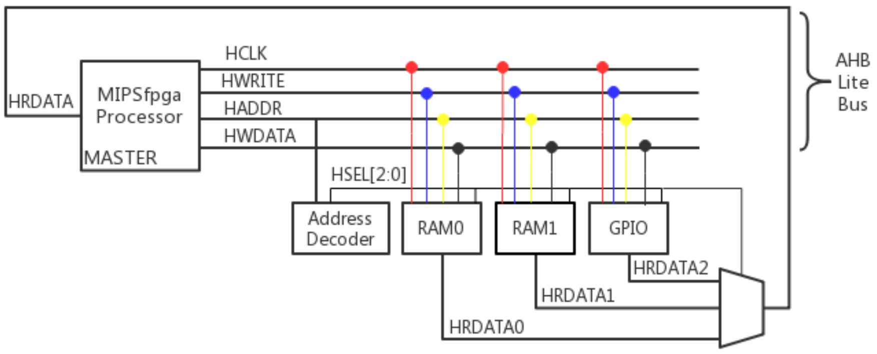


Figure 2.3: AHB-Lite Bus

The MIPSfpga processor core is the only MASTER connected to the AHB-Lite Bus. In addition, three slaves RAM0, RAM1 and GPIOs are connected to the MASTER. Memory block RAM0 contains the boot code while memory block RAM1 contains the user code and application data. During FPGA prototype the GPIO modules implement the connections to the LEDs, switches, pushbuttons and the 7-segmemt displays available on the used Nexys4 DDR board. For ASIC implementation, the RAM0 and RAM1 modules have to be adapted to the SRAM modules provided by TSMC before functional simulation which is described in chapter 5.

In addition to the three slaves, an address decoder and a data multiplexer are also available to generate the selection signals *HSEL[2:0]* based on the *HADDR* address and decide which of the three slaves the read data bus *HRDATA* by means of a 3:1 multiplexer.

The corresponding virtual address of the RAM0 block holding the boot load is *0xbfc0_0000-0xbfc1_fffc* while the physical address is *0x1fc0_0000-1fc1_fffc*. As for the RAM1, which contains the user code, the virtual address and physical address is respectively *0x8000_0000-*

Virtual address	Physical address	Signal name	Nexys4 DDR
0xbf80_0000	0x1f80_0000	IO_LEDR	LEDs
0xbf80_0008	0x1f80_0008	IO_SW	switches
0xbf80_000c	0x1f80_000c	IO_PB	U, D, L, R, C push buttons
0xbf80_0010	0x1f80_0010	SEGEN_N[7:0]	AN[7:0]
0xbf80_0014	0x1f80_0014	SEG0_N[3:0]	Digit 0 value
0xbf80_0018	0x1f80_0018	SEG1_N[3:0]	Digit 1 value
0xbf80_001c	0x1f80_001c	SEG2_N[3:0]	Digit 2 value
0xbf80_0020	0x1f80_0020	SEG3_N[3:0]	Digit 3 value
0xbf80_0024	0x1f80_0024	SEG4_N[3:0]	Digit 4 value
0xbf80_0028	0x1f80_0028	SEG5_N[3:0]	Digit 5 value
0xbf80_002c	0x1f80_002c	SEG6_N[3:0]	Digit 6 value
0xbf80_0030	0x1f80_0030	SEG7_N[3:0]	Digit 7 value

Table 2.2: Memory addresses used during FPGA prototype on the Nexys4 DDR FPGA board

0x8003_fffc and *0x0000_0000-0003_fffc*. The LEDs, switches, pushbuttons, the 7-segment displays, and enable signals for each digit are mapped to virtual memory addresses *0xbf80_0000-0xbf80_0030*, as shown in Table 2.2. The MMU on the MIPSfpga translates virtual addresses used by the processor core into physical addresses, received by the AHB-Lite Bus.

2.4.1 Connection between each component of the AHB-Lite Bus

The implementation schematic of the AHB-Lite bus components is shown in Figure 2.4. The address register adrreg sends the address from the address signal bus *HADDR [31:0]* to the ahbdecoder and to the memory block RAM0 and RAM1. The component writes data to the register writereg and forwards the write enable signal from *HWRITE* to the slaves.

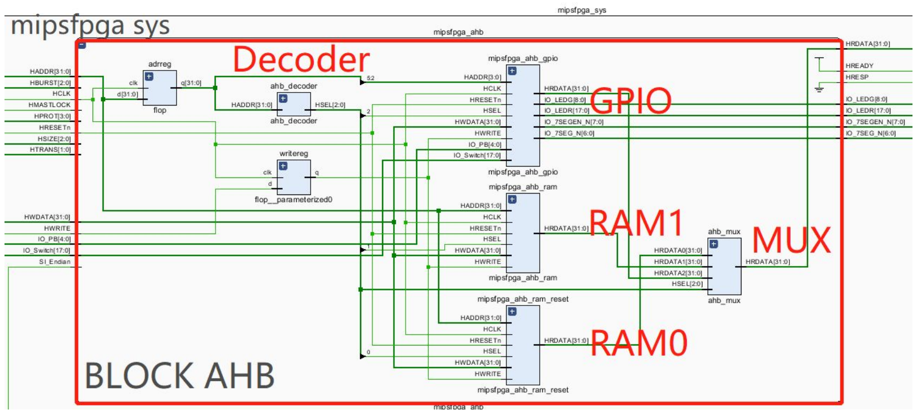


Figure 2.4: AHB components

```
module ahb_decoder
(
    input  [31:0] HADDR,
    output [ 2:0] HSEL
);
//H_RAM_RESET_ADDR_Match=7'h7f;
//H_RAM_ADDR_Match=1'b0;
//H_LEDR_ADDR_Match=7'h7e;
// Decode based on most significant bits of the address
assign HSEL[0] = (HADDR[28:22] ==`H_RAM_RESET_ADDR_Match);
assign HSEL[1] = (HADDR[28]     == `H_RAM_ADDR_Match);
assign HSEL[2] = (HADDR[28:22]  == `H_LEDR_ADDR_Match);
endmodule
```

Listing 1: Verilog Code for AHB Decoder

As shown in Listing 1, the bits of the selection signal *HSEL [2]*, *HSEL [0]* depend on the bits in *HADDR [28:22]* and *HSEL [1]* is derived from the *HADDR [28]* bit. The output *HRDATA [31:0]* from the component GPIO contains the data from the peripheral, which is named as *HRDATA2 [31:0]* in the Verilog module *mipsfpga_ahb.v* and is selected when the signal *HSEL [2:0]* equals 3'b100. The *HRDATA [31:0]* from RAM0 contains the machine codes, which is named as *HRDATA0 [31:0]* in the Verilog module *mipsfpga_ahb.v* and is selected when *HSEL[0]* equals 1'b1. The read data bus *HRDATA [31:0]* and the signals *HREADY*, and *HRESP* of the AHB-Lite bus block are fed to the block top, which corresponds to the Verilog module *m14k_top.v*. The other outputs of the AHB-Lite bus block are used as outputs of the top-level hierarchy module *mipsfpga_sys.v*, as shown in the following Figure 2.5.

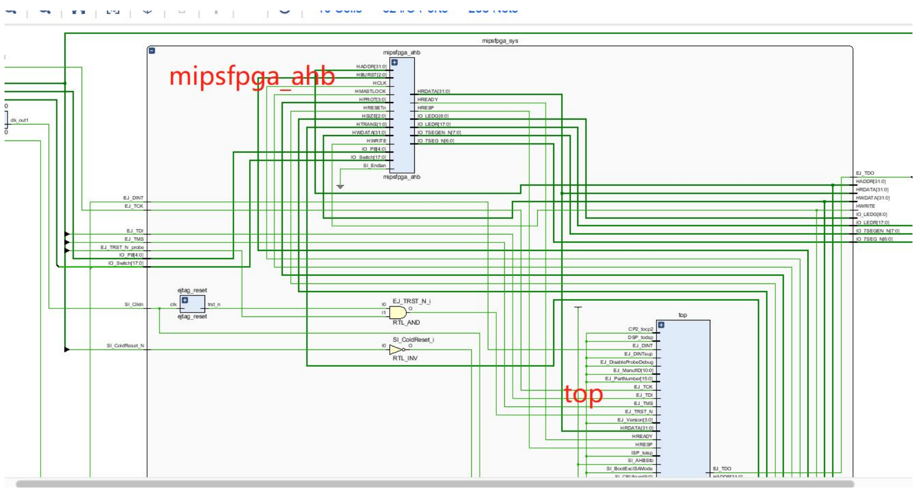


Figure 2.5: Connection between the block AHB-Lite and Top

2.5 Required Logic Blocks

The microAptiv™ UP core consists of both required and optional blocks. As shown in the block diagram in Figure 2.6, the frames shown in white of the block diagram are mandatory blocks and are always used for the proper execution of instructions. The areas shown in grey of the block diagram are optional blocks. The use of these blocks relies on the needs of the specific implementation targeting a respective application. The following subsections introduce the mandatory blocks of the microAptiv UP processor core and their function.

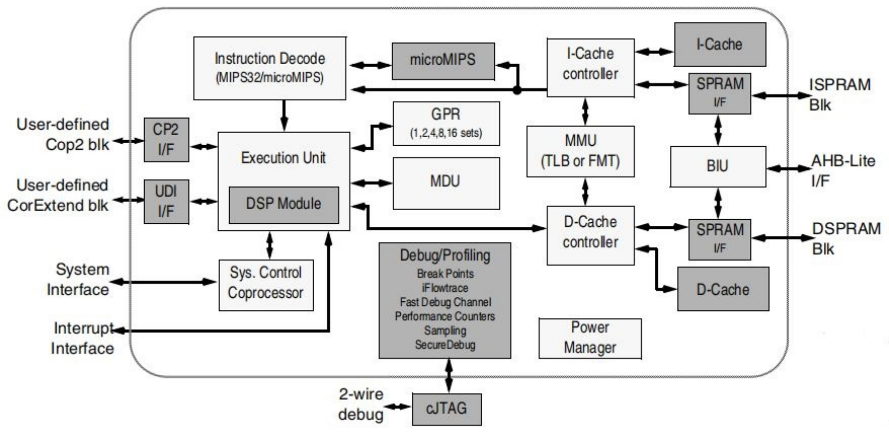


Figure 2.6: Mandatory and optional blocks in microAptiv UP core [1]

2.5.1 Bus Interface Unit

The bus interface unit *m14k_biu* acts as the interface between the microAptiv UP processor core and the outside blocks. Read/write requests from the cache controller are transmitted to the BIU. The requests are arbitrated and transformed to bus transactions according to the AMBA-3 AHB-lite protocol [1].

2.5.2 Cache Controller

The caches sizes, organizations, and set-associativity depends on the microAptiv UP core configuration and the deployed data controllers. For example, the size of the data cache can be 2 Kbytes, and the set-associative of the data cache is defined as 2-way, whereas the size of the instruction cache can be 8 Kbytes and the set-associative of the instruction cache is defined as 4-way. The CPU core can reach each cache in a single processor cycle. Also, each cache has its 32-bit data path, and the core can access both caches in the same pipeline clock cycle. A one-line fill buffer is included in each cache controller and managed accordingly. The fill buffer collects the data to be written to the cache and can be accessed in parallel to the cache. The data can be bypassed and written back to the core [1]. After the virtual-to-physical translation in the *m14k_mmu* module, the most significant bits [31:10] of the instruction physical address is merged with parts of the offset [9:2], originating from the execution module *m14k_edp*, in the instruction cache controller *m14k_icc*, and the addressed instructions are read from the AHB module and transmitted through the cache control module. In the data cache controller *m14k_dcc*, the physical data address from the instructions like *lw* or *sw* is processed in the same way. However, the offset is defined by another signal which also originates from the *m14k_edp* module. Addressed data is transferred from the memory block RAM0 through the data cache controller module to the core. The exact process for both modules is introduced in chapter 4.

2.5.3 Master Pipeline Control

The master pipeline control module *m14k_mpc* receives the machine code from the instruction cache controller and sends it to the block decoder *m14k_mpc_dec* in the MPC. This block decodes the instructions and extracts the register addresses of the operands, and forwards them to the general-purpose-register (GPR) module. The block *m14k_mpc_ctl* generates most of the control signals which are used during the execution stage in the module MPC.

2.5.4 Execution Data Path

The execution data path (*m14k_edp*) module is the execution unit of the CPU. On reset, the processor begins in kernel mode and jumps to the reset vector at address *0xbfc00000* which is the first generated virtual address after reset in the execution data path module. In this module the virtual address which points to the next instruction to be fetched is also calculated. Basically, the module executes the instructions which have been received from the *m14k_mpc* module. It receives the data for load instructions asserted on the read data signal *HRDATA* of the AHB-Lite bus interface. It also generates the data for the store instructions, which is set to the write data signal *HWDATA*. The function of the implemented modules like the ALU is described as a set of several assignments in the Verilog code.

2.5.5 Memory Management Unit

The MMU in the microAptiv UP processor core translates virtual addresses to physical addresses before request are sent to the cache controllers for tag comparison or to the bus interface unit for an external memory reference. This translation is a feature for operating systems which manage the physical memory in a way that it accommodates multiple tasks active in the same memory, so that they operate on the same virtual address space but in different locations in physical memory. Other features handled by the MMU are protected memory areas and the definition of cache protocols [1]. The MMU is translation lookaside buffer (TLB) based by default and has three address translation buffers:

- 16 or 32 dual-entry full associate Joint TLB(JTLB)
- 4-entry fully associate Instruction micro TLB(ITLB)
- 4-entry fully associate Data micro TLB(DTLB)

In the following, these three buffers are introduced one after another. A 16 or 32 dual- entry, fully associative Joint TLB, implemented by the microAptiv UP processor core, maps 32 or 64 virtual pages to their corresponding physical addresses. The upper bits of the virtual address are compared with each of the entries in the tag portion of the JTLB structure to translate virtual addresses and the corresponding Address Space Identifier (ASID) into physical addresses. Since both instruction and data virtual addresses are translated in the same way the structure is called "Joint" TLB.
In order to reduce the overall size, the JTLB is formed in page entry pairs. Two corresponding physical data entries of each virtual tag form the even odd page entry. The highest order virtual address bit is used for the selection of the two data entries. Because of the page size, the page-pair basis is variable. The bit for the even/odd selection and the address bit for the comparison are not the same during TLB lookup. Figure 2.7 shows the contents of one of

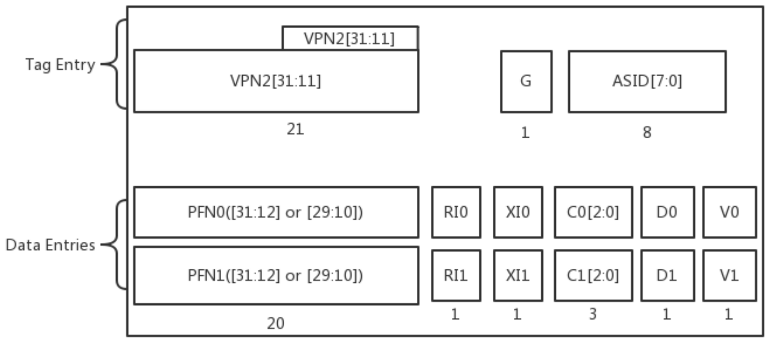


Figure 2.7: JTLB Entry (tag and data) [1]

the dual-entries in the JTLB. The tag entry consists of the Virtual Page Number divided by 2 (VPN2). Bits 31:25 are always included in the TLB lookup comparison. Bit 24:13 are set by the page mask value, which defines the page size by masking the proper VPN2 bits. The page mask value also decides which bits determine the selection of the even-odd page frame number(PFN0-PFN1). As for the 4KB page size, the virtual address bit [12] determines the page choice. The second part of this entry is the global bit. The entry is global to the whole processor when G is set. Furthermore, it also means that the ASID is not included in the comparison. The last 8 bits form the AISD. The address-space for each process is associated with the TLB entries through ASID. PFN0 and PFN1 are the upper bits of the physical address. The

Name	Number	Use
$0	0	the constant value 0
$at	1	assumbly temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure arguments
$t0-$t7	8-15	temporary variables
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	temporary variables
$k0-$k1	26-27	operating system (OS) temporaries
$gp1	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

Table 2.3: MIPS register set

remaining data entries are not described in detail. The ITLB performs the memory address translation for the instruction. When the ITLB cannot translate the address, the JTLB tries to finish translating of the address in the following clock cycle. The translated address is copied into the ITLB after successful execution, and the ITLB can resume operation again [1]. Unlike the ITLB, the DTLB applies a faster translation for Load/Store addresses which works in parallel to the JTLB. A DTLB miss and a JTLB hit effects a DTLB reload in the same cycle. The process of the virtual-to-physical address translaion is described in more detail in chapter 3.

2.6 Register Set

The MIPS architecture contains 32 registers (*$0, $1,..., $31*), the program counter (pc), and two special registers *Hi* and *Lo* to store the result of multiplications and divisions. These two special registers cannot be used directly, but they are only accessible through two instructions *mfhi* (move from *Hi*) and *mflo* (move from *Lo*). The results of multiplications and divisions can be longer than 32 bits. With multiplications the most significant bits are placed in the Hi register, and the least significant bits are placed in the *Lo* register. Division operations put the quotient in the *Lo* and the remainder in the *Hi* register.

The first register *$0* contains the constant value 0. Since this register is read-only, it should be used only as a source operand and is ineffective as the destination operand. The return address of a function or service routine should be stored in the last register *$ra.* The other 30 registers and the convention regarding their usage are listed in Table 2.3.

2.7 INSTRUCTION SET

The Instruction set of the MIPS architecture follows the RISC (Reduced Instruction Set Computer) principle. RISC computer use only simple instruction with fewer cycles per instruction (CPI) than a CISC (Complex Instruction Set Computer). In this section, the function and the format of the instruction set is explained.

2.7.1 FUNCTIONAL INSTRUCTIONS

Instructions have four different kinds of functional groups:

- CPU Load & Store Instructions.
- Arithmetic Logic Operation (add, sub, and, or...).
- Jump & Branch.

2.7.1.1 CPU Load and Store Instructions

All instructions have a width of 32 bits. Data is organized in words of 4 Bytes (32 Bit), half-words of 2 Bytes (16 Bit), and bytes (8 Bit) which can be stored from registers to memory or oppositely be loaded from memory to registers. Memory is accessible through dedicated load and store instructions, such as *lw*, *sw*. The address consists of two parts. The first part of the address is stored in a source register "*rs*", the second part is given as constant offset in the instruction. The addressed data will be stored in a destination register "*rd*" . For example, the *LHU/LH* instruction loads a half-word either unsigned by filling the most significant bits with zeros or sign-extended by filling the most significant bits with the respective sign bits. A list of load store instructions is given in Table 2.4.

Mnemonic	Instruction	Function
lb rd, of (rs)	Load Byte	rd=mem[rs+of]
lbu rd, of (rs)	Load Byte Unsigned	rd=mem[rs+of]
lh rd, of (rs)	Load Halfword	rd= mem[rs+of]
lhu rd, of (rs)	Load Halfword Unsigned	rd= mem[rs+of]
lw rd, of (rs)	Load Word	rd= mem[rs+of]
sb rs, of (rt)	Store Byte	mem[rt+of] = rs[7:0]
sh rs, of (rt)	Store Halfword	mem[rt+of] = rs[15:0]
sw rs, of (rt)	Store Word	mem[rt+of] = rs

Table 2.4: Load and Store instructions

2.7.1.2 Arithmetic and Logic Operations

The MIPS architecture has different kinds of arithmetic instructions, including *add*, *sub*, *div*, *mult*. Some instructions come with an "*i*" or an "*u*" are at the end of the mnemonic code to indicate special modes. "*i*" means that the respective instruction is used with an immediate which is a 16-bit constant operator in combination with data stored in the source register "*rs*". "*u*" indicates an unsigned instruction. The rest of the instructions use the operator *rt* (register target) and *rs* (register source), which can be dynamically changed in the program. The results are stored in the destination register (*rd*) after the execution of every instruction. Attention should be paid to division and multiplication instructions, which use specific register lo and hi to store the results as mentioned above.

Mnemonic	Instruction	Function
add rd, rs, rt	adds two registers and store result in a register	rd= rs+rt
addi rd, rs, im	adds a register and a sign-extended immediate and store result in a register	rd= rs+im
addu rd, rs, rt	adds two registers and stores the result in a register unsigned	rd= rs+rt

div rs, rt	divides rs by rt and stores the quotient in lo and the remainder in hi	lo=rs/rt
		hi=rs%rt

sub rd, rs, rt	substrates two registers and stores result in a register	rd= rs-rt
subu rd, rs, rt	substrates two registers and stores result in a register unsigned	rd= rs-rt

mult rs, rt	multiplies rs by rt and stores the result in lo	hi=(rs*rt)[63:32]
		lo=(rs*rt)[31:0]

Table 2.5: Load and Store instructions

Logical operation like "and", "or", "xor", and "nor" operate bit-by-bit on two source registers. The result is written in the destination register "*rd*". However, a "Not" instruction does not exist. Instead, the "Not" instruction is implemented by using a "Nor" instruction in combination with the zero register *$0*. As shown in Table 2.5, logical operations can also be operated on immediates using instructions "*andi*", "*ori*", "*xori*".

Mnemonic	Instruction	Function
and rd, rs, rt	Ands bitwise two registers and stores the result in a register	rd=rs & rt
andi rd, rs, im	ands bitwise a register and an immediate value and stores the result in a register	rd=rs & im
or rd, rs, rt	Ors two registers bitwise logical and stores the result in a register	rd=rs | rt
ori rd, rs, im	Ors a register and an immediate value bitwise and stores the result in a register	rd= rs | im
xor rd, rs, rt	Exclusive or of two registers. Results are stored in a register	rd= rs ∧ rt
xori rd, rs, im	Bitwise exclusive or of a register and an immediate value. Results is stored in a register	rd= rs ∧ im

Table 2.6: Logical operations

Shift Instructions can move each digit in a register left or right by the amount of bits given in the parameter "*shamt*". Normal shift instructions are "*sll*" (shift left logical), "*srl*" (shift right logical), and "*sra*" (shift right arithmetic). Shift right arithmetic "*sra*" means that the most significant bits are not filled with zero but with the sign bit of the initial value. A mnemonic code that ends on "v", such as "*sllv*" (shift left logical variable), "*srlv*" (shift right logical variable), "*srav*" (shift right arithmetic variable) means that the amount of bit positions to be shifted is defined dynamically in a register.

Mnemonic	Instruction	Function
sll rd, rs, a	Shifts a register value left by the amount of bits defined in the instruction and stores the result in register rd	rd = rs «a
sllv rd, rs, rt	Shifts a register value left by the amount of bits defined in the register rt and stores the result in register rd	rd = rs «rt
sra rd, rs, a	Shifts a register value right by the amount of bits defined in the instruction and stores the result in register rd	rd = rs »a
srav rd, rs, rt	Shifts a register value right by the amounts of bits defined in the register rt and stores the result in register rd	rd = rs »rt
srl rd, rs, a	Shifts a register value right by the amount of bits defined in the instruction and stores the value in register rd	rd = rs »a
srlv rd, rs, rt	Shifts a register value right by the amount of bits defined in register rt and stores the value in register rd	rd = rs »rt

Table 2.7: Shift operations

2.7.1.3 Jump and Branch Instructions

The PC (Program Counter) is the pointer holding the memory address of the present instruction and is updated to point to the next sequential instruction to execute. Conditional Control Instructions (branch) change the PC based on the evaluation of conditions. Unconditional Control Instructions (jump) always change the PC. The field in the branch instructions that specifies the new instruction address is 16 bits wide and is given relative to the next (PC+4) and not the current instruction address (PC) when the branch instruction is executed. In jump instructions the size of the offset field is 26 bits. The offset is shifted left by two bits and filled up with the four most significant bits of the PC.

Mnemonic	Instruction	Function
beq rs, rt, of	branch on equal	pc = pc + (of «2)
bne rs, rt, of	branch on not equal	pc = pc + (of »2)
bgez rs, of	branch on greater than or equal zero	pc = pc + (of «2)
bgezal rs, of	branch on greater than or equal to zero and link	pc = pc + (of «2) $ra= pc + 8
bgtz rs, of	branch on greater than zero	pc = pc + (of »2)
blez rs, of	branch on less than or equal to zero	pc = pc + (of »2)
bltz rs, of	branch on less than zero	pc = pc + (of »2)
bltzal rs, of	branch on less than zero and link	pc = pc + (of »2), $ra= pc + 8
j ad	jump	pc = pc[31:30] & (ad «2)
jal ad	jump and link	pc= pc[31:30] & (ad «2), $ra= pc +8
jalr rd, rs	jump and link register	pc= rs, $ra= pc + 8

Table 2.8: Shift operations

2.7.2 Instructions Format

Instructions are written in binary code format and consist of a string of 1′s and 0′s. To identify the instructions, the 32-bit binary code is divided into several fields to represent different pieces of information. There are three different instruction formats: R-Type, I-Type, and J-Type.

2.7.2.1 R-Type Instruction

R-Type instructions are short for register type and operate on three registers. *rs* and *rt* represent the source register and *rd* is the destination register. Table 2.9 shows the R-Type machine instruction format.

The *op* field of R-Type-instructions is always filled with zeros, while the *funct* field determines the operation. The *shamt* field is used in shift instruction to define the amount of bits to shift.

For example, the add instruction is defined by the code 32 (100000_2) and 0 (000000_2) in the opcode and the funct fields, respectively.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Table 2.9: R-Type

rs, *rt* are source registers, *rd* is the destination register. The register numbers can be found in Table 2.1. Table 2.10 shows the machine code for the R-type instructions add and sub.

Assembly Code	Field Values					
	op	rs	rt	rd	shamt	funct
add \$s0, \$s1, \$s2	0	17	18	16	0	32
sub \$t0, \$t3, \$t5	0	11	13	8	0	34
Machine Code						
	000000	10001	10010	10000	00000	100000
	000000	01011	01101	01000	00000	100010

Table 2.10: R-Type examples

2.7.2.2 I-Type Instruction

The I-Type is short for immediate-type, and consists of two register operands and a 16-bit “immediate” value. The naming of the instruction format indicates that it is much faster to access constants given in the instruction than other available data sources. The 32-bit instruction has four fields: *op*, *rs*, *rt*, and *imm*. Table 2.11 shows the I-Type machine instruction format.

The *opcode* determines the operation of I-Type. *rs* and *imm* are used as source operands. For some instructions (*addi* and *lw*) *rt* is used as the destination, while for others instructions it is used as another source register. Table 2.12 shows the machine code for the I-Type instructions *addi* and *sw*.

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Table 2.11: I-Type

Assembly Code	Field Values			
	op	rs	rt	imm
addi $s0, $s1, 5	8	17	16	5
sw $s1, 4($t1)	43	9	17	4
Machine Code				
	001000	10001	10000	0000000000000101
	101011	01001	10001	0000000000000100

Table 2.12: I-Type examples

2.7.2.3 J-Type Instruction

J-Type is short for jump-type and is only used with jump instructions. This instruction consists of an *opcode* and the 26-bit address operand, *addr*. Table 2.13 shows the J-Type machine instruction format.

op	addr
6 bits	26bits

Table 2.13: J-Type

3 VIRTUAL-TO-PHYSICAL ADDRESS TRANSLATION IN MMU

The translation from a virtual address to a physical address happens in the MMU, an interface between the execution unit and the cache controller. The microAptiv UP core contains a Translation Lookaside Buffer (TLB) or a simple Fixed Mapping Translation (FMT) style MMU [1]. In this chapter, both approaches are described. In addition, one more special translation method exists for unmapped segments.

3.1 VIRTUAL-TO-PHYSICAL ADDRESS TRANSLATION FOR UNMAPPED SEGMENTS

In this thesis, all physical addresses stay in the unmapped segment kseg1. Because the core enters Kernel mode both at reset and when an exception is recognized, an unmapped segment does not use the TLB or the FMT approach for the virtual-to-physical address translation. Especially after reset, the availability of unmapped memory segments is essential because the TLB is not yet programmed to perform the translation[1]. The processor operates in Kernel mode when the *DM* bit in the Debug register is 0, and one or more of the following three bits in the Status register are set as follows *UM*=0, *ERL*=1, *EXL*=1. When a non-debug exception is detected, *EXL* or *ERL* is set, and the processor enters Kernel mode.
In Kernel mode, when the most-significant three bits of the 32-bit virtual address is 0xb101,

the 32-bit kseg1 virtual address space is selected. kseg1 is the 2^{29} byte (32-MByte) kernel virtual address space located at address *0xA000_0000 - 0xBFFF_FFFF*. References to kseg1 are unmapped, the selected physical address is defined by subtracting *0xA000_0000* from the virtual address. In addition the caches are disabled for accesses to these addresses, and the physical memory (or memory-mapped I/O device registers) are accessed directly [1].
The algorithmic translation process in the Verilog code is different from the subtraction mentioned above and is described in the next chapter. As mentioned before the virtual addresses are *0xBFC0_0000* or *0xBFC0_0008*. The binary code for the hexadecimal number B is 4'b1011, and the upper three bits are identical to the condition. Figure 3.1 shows three important bits for the determination of the kernel mode.

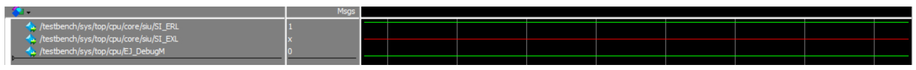

```
                                              Msgs
/testbench/sys/top/cpu/core/siu/SI_ERL        1
/testbench/sys/top/cpu/core/siu/SI_EXL        x
/testbench/sys/top/cpu/EJ_DebugM              0
```

Figure 3.1: Bit ERL,EXL,DM

3.2 Virtual-to-Physical Address Translation in TLB based MMU

In this subsection, the translation in TLB based memory management is discussed. Memory translation happens in the MMU module of the micorAptiv UP processor core, the virtual-to-physical address translation is implemented by the method introduced in the first section of this chapter. Except for kseg0 and kseg1 for all other segments that are mapped or cached TLB can be used.

When the translation process is activated, the virtual address from the processor can be the same or different in comparison with the virtual address in the TLB. The situation where both addresses are identical is called match or hit, and the first condition is that the VPN of the address is the same as the VPN field of the entry. In addition, the two following conditions are checked.

- The Global bit of both the even and odd pages of the TLB entry is set.
- The ASID field of the virtual address is the same as the ASID field of the TLB entry.

If all these conditions are not matched, the processor starts a TLB miss exception, and the software refills the TLB from a page table of virtual/physical addresses in memory. As shown in Figure 3.2, the page frame number (PFN) of the physical address comes from the PFN in the TLB, if the virtual address is matched in the TLB. The offset does not pass through the TLB and is connected with the PFN to form the physical address.

3.3 Virtual-to-Physical Address Translation in FMT based MMU

An optional unit for the microAptiv UP core is the Fixed Mapping memory management unit that is smaller than a full TLB and synthesized with less hardware effort. Unmapped memory segments in a TLB implementation (kseg0 and kseg1) are translated identically by the FMT

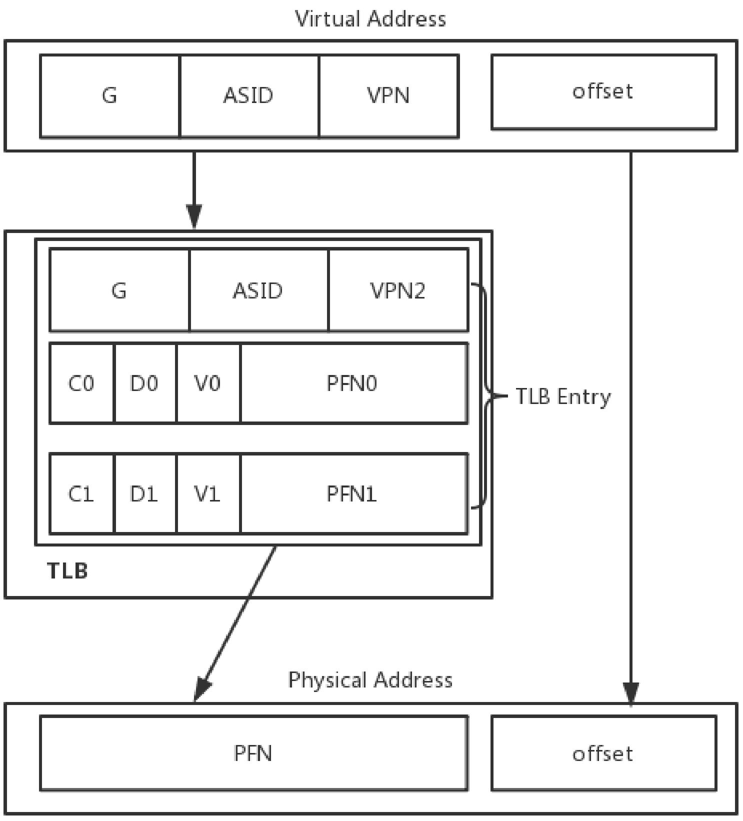


Figure 3.2: Virtual-to-Physical address translation in TLB base MMU

MMU, and the translation result could be explained by using the same method as mentioned in the first section for FMT. However, the applied process of both methods is not the same [1]. With the FMT approach, a bitwise AND of the virtual address space with a fixed number of *0x1FFF_FFFF* is performed which means that the translated result of the virtual address *0xBF80_0004* is the physical address *0x1F80_0004* [3].

4 Execution Process of The MicroAptiv™ Up Processor Core

This chapter describes the execution process of the microAptiv up processor core. The first section starts with a brief description of how mnemonic MIPS assembly code is translated to machine code, on which the simulation results in this section are based. The simulation results generated by Modelsim are used to analyze the core's operational logic. This section also introduces the requirements of the following two chapters with respect to functional and power simulations. In addition, the generation of the first physical address and the increment of the addresses are described. One method for virtual to physical address translation through different processes in Verilog code is fixed simple translation, which has already been analyzed in section 3.1. The pipeline stage for the *lw* instruction will be introduced in the third section. Because of the different transformation steps from virtual to physical addresses during the execution of *lw* and *sw* instructions, both processes are introduced in the fourth section. The last section will describe the Write signal *HWDATA* and the Read data process from the AHB bus to the processor core. All connections of the following signals can be found in the Verilog code. In each diagram and simulation results, the main signals are shown. The following provides a detailed understanding of how addresses and data are propagated within the processor, which is particularly essential for the verification process described in Chapter 6 to detect undefined signals generated during the gate-level simulation and to fix the issues causing them.

4.1 Text File Creation for the Initialization of MIPSfpga Memories

The CodeScape MIPS SDK Essentials is available for software programming in C and Assembly. The CodeScape package includes an installation of the OpenOCD software for on-chip debugging. For compilation or assembly, the assembly programs and the Makefile from the directory Codescape\ExamplePrograms\Assembly of the MIPSfpga project should be copied to the directory in which the compile or assemble file main.c or main.s can be found. For the compilation, the command make is used. By this means the file FPGA_Ram.elf in ELF (Executable and Link) format is generated. The text files which initialize the MIPSfpga memories are called *ram_reset_init.txt* and *ram_program_init.txt*. These memory initialization files should be created to run a program during simulation with Modelsim and Xcelium. For this purpose, the command shell *cmd.exe* is opened, and the user has to change to the respective directory.

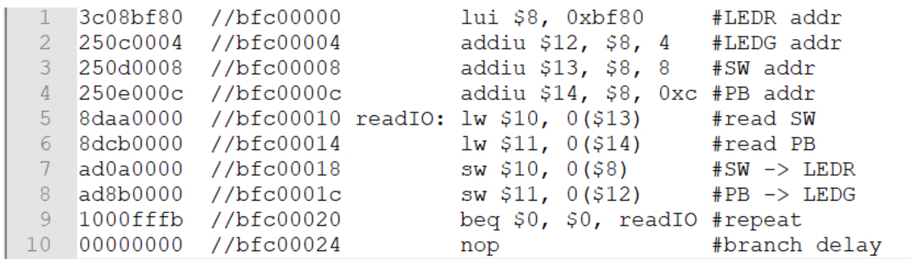
```
cd C:\...\MIPSfpga\Codescape\ExamplePrograms\Scripts
```

Then, the memory files are generated by typing the following command into the command prompt:

```
createMemfiles.bat ...\Examplefolder
```

The generated text files *ram_reset_init.txt* and *ram_program_init.txt* contain the machine instructions based on the compiled code in the Example folder in a format that allows the block RAM initialization (i.e, Codescape\ExamplePrograms\Examplefolder). Moreover, a MemoryFiles folder is created that contains the memory initialization files. In order to enable simulation, the machine code, which start at @1D7 and contains the same amount of lines as the initial program, needs to be copied from the initial text file *ram_program_init.txt* to the file *ram_reset_init.txt*. These files can be used in Modelsim to simulate and study the processor execution process. However, this initial data does not match the format in TSMC memory, which is described in detail in Chapter 5.

- **Assembly Program**
  The following program is used for generating the initial machine code file. The purpose of this code is to allow a clear analysis of how the store/load instructions (*sw*, *lw*) are processed in the core, to understand the operating principle of the entire system. The program first writes the addresses of the switches and pushbuttons into registers. By using the *lw* instruction the data is read from the switches into registers. Through the *sw* instruction the data is stored into the virtual address where the led register is located. Finally, this process is looped through the *beq* instruction.



```
3c08bf80  //bfc00000         lui $8, 0xbf80     #LEDR addr
250c0004  //bfc00004         addiu $12, $8, 4   #LEDG addr
250d0008  //bfc00008         addiu $13, $8, 8   #SW addr
250e000c  //bfc0000c         addiu $14, $8, 0xc #PB addr
8daa0000  //bfc00010 readIO: lw $10, 0($13)     #read SW
8dcb0000  //bfc00014         lw $11, 0($14)     #read PB
ad0a0000  //bfc00018         sw $10, 0($8)      #SW -> LEDR
ad8b0000  //bfc0001c         sw $11, 0($12)     #PB -> LEDG
1000fffb  //bfc00020         beq $0, $0, readIO #repeat
00000000  //bfc00024         nop                #branch delay
```

Figure 4.1: Assemble Program for the lw and sw instructions

- **Initial Files**
  The following is the initial machine code which corresponds to the program shown above and is used in the simulation. In order to verify that the machine code and the MIPS assembly program are identical, the instruction like *lw* can be checked through the I-type instruction format as is shown in table 4.1. The table below is the machine code for the first *lw* instructions.

```
3c08 bf80
250c 0004
250d 0008
250e 000c
8daa 0000
8dcb 0000
ad0a 0000
ad8b 0000
1000 fffb
```

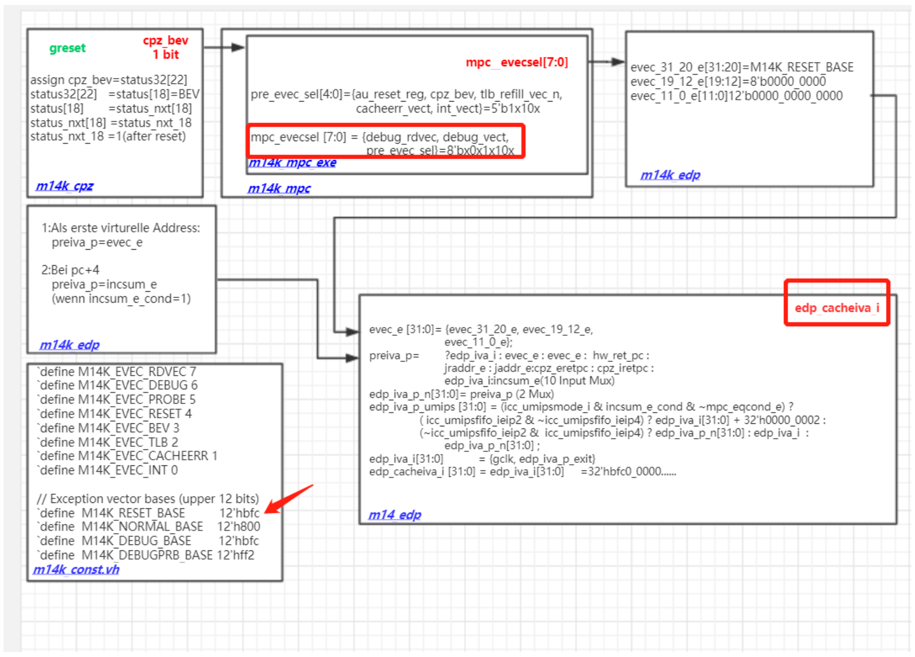
Listing 2: Initial machine code

Assembly Code	Field Values			
	op	rs	rt	imm
lw $10, 0($13)	35	13	10	0
Machine Code				
	100011	01101	01010	0000000000000000

Table 4.1: Machine code for first lw instruction

4.2 Initialization of The Processor Core

Figure 4.2 illustrates the generation process of the signal *edp_cacheiva_i*. In each rectangular frame, there are short descriptions of the primary signal transmission process. In the module some signals and parameters defined in the *m14k_const.vh* file are used in this initialization process.



```
assign cpz_bev=status32[22]
status32[22]   =status[18]=BEV
status[18]       =status_nxt[18]
status_nxt[18] =status_nxt_18
status_nxt_18 =1(after reset)
```

```
pre_evec_sel[4:0]={au_reset_reg, cpz_bev, tlb_refill_vec_n,
                   cacheerr_vect, int_vect}=5'b1x10x

mpc_evecsel [7:0] = {debug_rdvec, debug_vect,
                     pre_evec_sel}=8'bx0x1x10x
```

```
evec_31_20_e[31:20]=M14K_RESET_BASE
evec_19_12_e[19:12]=8'b0000_0000
evec_11_0_e[11:0]12'b0000_0000_0000
```

```
1:Als erste virturelle Address:
  preiva_p=evec_e

2:Bei pc+4
  preiva_p=incsum_e
  (wenn incsum_e_cond=1)
```

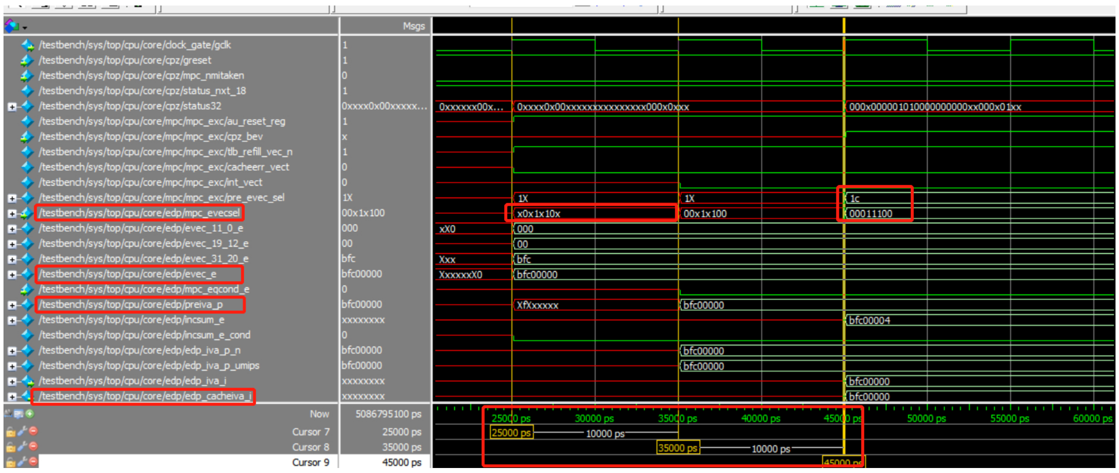
```
evec_e [31:0]= {evec_31_20_e, evec_19_12_e,
               evec_11_0_e};
preiva_p=      ?edp_iva_i : evec_e : evec_e :  hw_ret_pc :
               jraddr_e : jaddr_e:cpz_eretpc : cpz_iretpc :
               edp_iva_i:incsum_e(10 Input Mux)
edp_iva_p_n[31:0]= preiva_p (2 Mux)
edp_iva_p_umips [31:0] = (icc_umipsmode_i & incsum_e_cond & ~mpc_eqcond_e) ?
           ( icc_umipsfifo_ieip2 & ~icc_umipsfifo_ieip4) ? edp_iva_i[31:0] + 32'h0000_0002 :
           (~icc_umipsfifo_ieip2 &  icc_umipsfifo_ieip4) ? edp_iva_p_n[31:0] : edp_iva_i  :
               edp_iva_p_n[31:0] ;
edp_iva_i[31:0]         = {gclk, edp_iva_p_exit}
edp_cacheiva_i [31:0] = edp_iva_i[31:0]     =32'hbfc0_0000......
```

```
`define M14K_EVEC_RDVEC 7
`define M14K_EVEC_DEBUG 6
`define M14K_EVEC_PROBE 5
`define M14K_EVEC_RESET 4
`define M14K_EVEC_BEV 3
`define M14K_EVEC_TLB 2
`define M14K_EVEC_CACHEERR 1
`define M14K_EVEC_INT 0

// Exception vector bases (upper 12 bits)
`define  M14K_RESET_BASE        12'hbfc
`define  M14K_NORMAL_BASE      12'h800
`define  M14K_DEBUG_BASE        12'hbfc
`define  M14K_DEBUGPRB_BASE 12'hff2
```

Figure 4.2: Diagram illustrating the calculation of the first physical address 32'hbfc00000

As shown in Figure 4.3, after reset, the signal *au_reset_reg* in module *m14k_mpc_exe* is set to 1. After two clock cycles, the signal *cpz_bev* (bootstrap exception vector) is set to 1. Then, the processor works in kernel mode, which is unmapped and uncached. Both signals are derived from the control signal *mpc_evecsel=8'bx0x1010x* which is an input signal of module *m14k_edp*, and an output signal of the module m14k_mpc.
As shown in Figure 4.2, the signal *preiva_p* is assigned the value *32'hbfc00000* through the 10-to-1 multiplexer in the execution module *m14k_edp* based on the signal *evec_e* after one

clock cycle, which is controlled by the signal *mpc_evecsel* and mainly consists of the defined expression *'M14K_RESET_BASE=12'hbfc*. After one cycle the signal *edp_cacheiva_i* receives the data *32'hbfc00000* from the signal *preiva_p*.



```
                                                          Msgs
/testbench/sys/top/cpu/core/clock_gate/gclk               1
/testbench/sys/top/cpu/core/cpz/greset                    1
/testbench/sys/top/cpu/core/cpz/mpc_nmitaken              0
/testbench/sys/top/cpu/core/cpz/status_nxt_18             1
/testbench/sys/top/cpu/core/cpz/status32                  0xxxx0x00xxxxx...
/testbench/sys/top/cpu/core/mpc/mpc_exc/au_reset_reg      1
/testbench/sys/top/cpu/core/mpc/mpc_exc/cpz_bev           x
/testbench/sys/top/cpu/core/mpc/mpc_exc/tlb_refill_vec_n  1
/testbench/sys/top/cpu/core/mpc/mpc_exc/cacheerr_vect     0
/testbench/sys/top/cpu/core/mpc/mpc_exc/int_vect          0
/testbench/sys/top/cpu/core/mpc/mpc_exc/pre_evec_sel      1X
/testbench/sys/top/cpu/core/edp/mpc_evecsel               00x1x100
/testbench/sys/top/cpu/core/edp/evec_11_0_e               000
/testbench/sys/top/cpu/core/edp/evec_19_12_e              00
/testbench/sys/top/cpu/core/edp/evec_31_20_e              bfc
/testbench/sys/top/cpu/core/edp/evec_e                    bfc00000
/testbench/sys/top/cpu/core/edp/mpc_eqcond_e              0
/testbench/sys/top/cpu/core/edp/preiva_p                  bfc00000
/testbench/sys/top/cpu/core/edp/incsum_e                  xxxxxxxx
/testbench/sys/top/cpu/core/edp/incsum_e_cond             0
/testbench/sys/top/cpu/core/edp/edp_iva_p_n               bfc00000
/testbench/sys/top/cpu/core/edp/edp_iva_p_umips           bfc00000
/testbench/sys/top/cpu/core/edp/edp_iva_i                 xxxxxxxx
/testbench/sys/top/cpu/core/edp/edp_cacheiva_i            xxxxxxxx
                                                 Now      5086795100 ps
                                                 Cursor 7 25000 ps
                                                 Cursor 8 35000 ps
                                                 Cursor 9 45000 ps
```

Figure 4.3: Simulation of the signal edp_cacheiva_i

```
assign utlb_pah [`M14K_PAH] =
{PAHW{utlb_bypass}} & pre_pah |
{PAHW{~utlb_bypass & utlb0_match}} & utlb0_pah |
{PAHW{~utlb_bypass & utlb1_match}} & utlb1_pah |
{PAHW{~utlb_bypass & utlb2_match}} & utlb2_pah |
{PAHW{~utlb_bypass & utlb3_match}} & utlb3_pah;
```

Listing 3: Multiplexer for utlb_pah

As shown in Figure 4.4, the signal *edp_cacheiva_i=32'hbfc00000* is transformed into the 22 bit kseg virtual to physical mapping signal *i_kseg_addr=22'h07f000* through the control module *m14k_mmuc* in the MMU module.

signal	Binary / Hex	31_30	29_26	26_22	21_18	17_14	13_10
edp_cacheiva_i	2ff000	10	1111	1111	0000	0000	0000
cacheiva_trans	2ff000	10	1111	1111	0000	0000	0000
i_kseg_addr	07f000	00	0111	1111	0000	0000	0000

Table 4.2: Transformation from edp_cacheiva_i to i_kseg_addr

Table 4.2 shows that the signal *i_kseg_addr [31:29]* is transformed from 3'b101 to 3'b000, while the other bits do not change. The transformed data is sent to the signal *pre_ipah [31:10]*, and it is connected with the input port *pre_pah [31:10]* of module *m14k_tlb_itlb*. There are four entry blocks in this module with the same structure, but the output *utlb_pah* of the four

blocks is not selected.
As shown in Listing 3, the signal PHAW, short for PAH width is 22, and the signal *utlb_bypass* is 1. *PAHW{utlb_bypass}=22'h2FFFF* is bitwise ANDed with the signal *pre_pah* to get the same result stored in the signal *pre_pah*. The output *mmu_ipah [31:10]* receives the data from *pre_pah [31:10]*. As shown in Figure 4.5, *miss_tag_mx [31:10]* getssdf the data from *mmu_ipah*

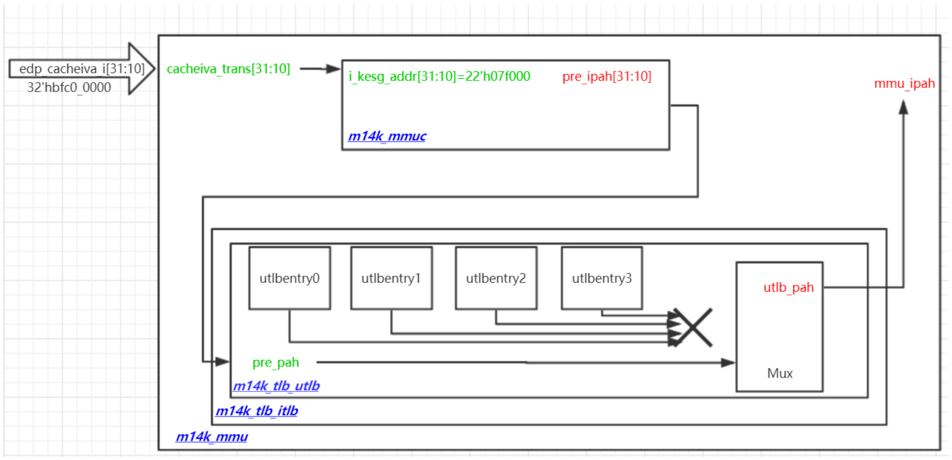


Figure 4.4: Diagram of the signal mmu_ipah[31:10]

[31:10]. It depends on the control signals *ld_mtag and icop_active_m* according to the simulation result in Figure 4.6. Table 4.3 shows that *miss_tag_mx [31:10]* and *miss_idx_mx [9:2]* are the two components of the *icc_exaddr [31:2]* signal.The signal *miss_tag_mx [31:10]* corresponds to the upper bits of the physical address, and the signal *miss_idx_mx [9:2]*, which originates from the signal *ival_i [9:2] = edp_ival_p [9:2]* which is the offset of the physical address.

signal	Binary / Hex	31_30	29_26	26_22	21_18	17_14	13_10	9_6	5_2
miss_tag_mx	07f000	00	0111	1111	0000	0000	0000		
miss_idx_mx	00							0000	0000
icc_exaddr	07f00000	00	0111	1111	0000	0000	0000	0000	0000

Table 4.3: icc_exaddr[31:2]={mixx_tag_mx[31:10],mixx_idx_mx[9:2]}

As shown in Figure 4.7, *iaddr[31:4]* gets parts of the virtual instruction address from *icc_exaddr [31:2]*, and the virtual address changes to *28'h1fc0_0000*. The combination of *iaddr [31:4]* and *iword_nxt [1:0]* is the main part of the signal *HADDR [31:2]*, which depends on the rising edge of the signal *ireq*. Another parts of the signal *HADDR [1:0]* is *be_nxt_address [1:0]*. At last *HADDR [31:0]* gets the first physical address *32'h1fc00000*. The generation of the next address depends on the rising edge of the signal *incsum_e_cond* and on the increment of *incsum_e*.

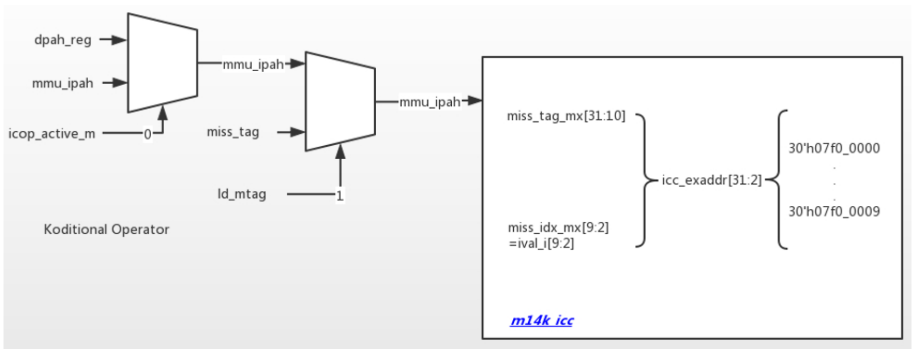
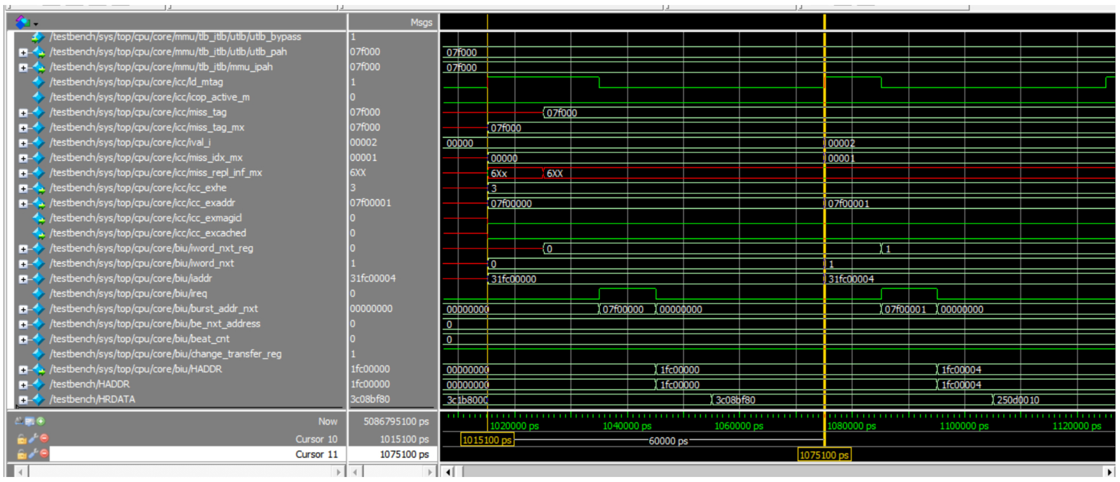


Figure 4.5: Diagram of the signal icc_exaddr[31:2]



	Msgs
/testbench/sys/top/cpu/core/mmu/tlb_itlb/utlb/utlb_bypass	1
/testbench/sys/top/cpu/core/mmu/tlb_itlb/utlb/utlb_pah	07f000
/testbench/sys/top/cpu/core/mmu/tlb_itlb/mmu_ipah	07f000
/testbench/sys/top/cpu/core/icc/ld_mtag	1
/testbench/sys/top/cpu/core/icc/icop_active_m	0
/testbench/sys/top/cpu/core/icc/miss_tag	07f000
/testbench/sys/top/cpu/core/icc/miss_tag_mx	07f000
/testbench/sys/top/cpu/core/icc/ival_i	00002
/testbench/sys/top/cpu/core/icc/miss_idx_mx	00001
/testbench/sys/top/cpu/core/icc/miss_repl_inf_mx	6XX
/testbench/sys/top/cpu/core/icc/icc_exhe	3
/testbench/sys/top/cpu/core/icc/icc_exaddr	07f00001
/testbench/sys/top/cpu/core/icc/icc_exmagicl	0
/testbench/sys/top/cpu/core/icc/icc_excached	0
/testbench/sys/top/cpu/core/biu/iword_nxt_reg	0
/testbench/sys/top/cpu/core/biu/iword_nxt	1
/testbench/sys/top/cpu/core/biu/iaddr	31fc00004
/testbench/sys/top/cpu/core/biu/ireq	0
/testbench/sys/top/cpu/core/biu/burst_addr_nxt	00000000
/testbench/sys/top/cpu/core/biu/be_nxt_address	0
/testbench/sys/top/cpu/core/biu/beat_cnt	0
/testbench/sys/top/cpu/core/biu/change_transfer_reg	1
/testbench/sys/top/cpu/core/biu/HADDR	1fc00000
/testbench/HADDR	1fc00000
/testbench/HRDATA	3c08bf80
Now	5086795100 ps
Cursor 10	1015100 ps
Cursor 11	1075100 ps

Figure 4.6: Simulation of the signal HADDR

4.3 Pipeline Stages

The execution pipeline consists of five stages:

- Fetch
- Decode
- Execute
- Memory
- Write back

This section presents the implementation of each stage in the microAptiv up processor core. As shown in Figure 4.8, the five stages and their main signals are described.

Each instruction follows a different process in these five stages.

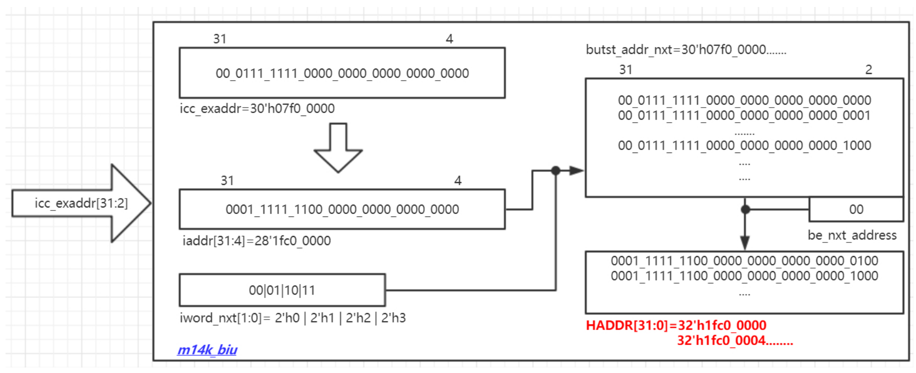


Figure 4.7: Diagram of the signal HADDR[31:0]

4.3.1 Pipeline Stages for lw Instruction

The last section describes the generation process of the first physical instruction address, which is sent to the *mipsfpga_ahb* module. The decoder in the *ahb_decoder* module and the multiplexer *ahb_mux* module select the respective slaves.
The Verilog code below shows the selection of each slave. When *HADDR [28:22]* equals to *H_RAM_RESET_ADDR_Match = 7'h7f = 7'b111_1111* and the other two conditions are not true, then *HSEL [2:0]* equals to 3'b001. Based on *HADDR [16:2]* applied to the *mipsfpga_ahb _ram_reset block*, the instructions are fetched sequentially and sent to the bus interface unit module *m14k_biu*.

```verilog
module ahb_mux
(
input        [ 2:0] HSEL,
input        [31:0] HRDATA2, HRDATA1, HRDATA0,
output reg   [31:0] HRDATA
);
always @(*)
casez (HSEL)
3'b??1:    HRDATA = HRDATA0;
3'b?10:    HRDATA = HRDATA1;
3'b100:    HRDATA = HRDATA2;
default:   HRDATA = HRDATA1;
endcase
endmodule
```

Listing 4: Verilog Code for AHB Selector

As shown in Figure 4.9, *HRDATA [31:0]* contains the instruction and is transmitted after one cycle clock through the multiplexer in module *m14k_biu*. The instruction cache controller module *m14k_icc* gets the output signal *biu_datain [31:0]* from the bus interface unit and transmits it through several blocks like *m14k_icc_umips_stub* and other multiplexers. The result of the process, as mentioned earlier, is *icc_idata_i [31:0]*. Next, the decode procedure is described. The decoding process is implemented in the module master pipeline control

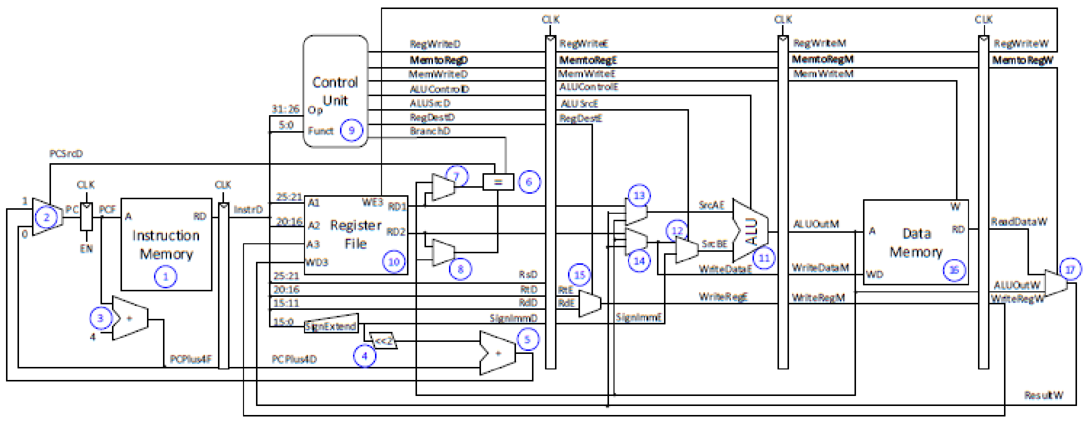


Figure 4.8: Diagram of the pipeline [4]

m14k_mpc. The *rs* field of the instruction is *icc_idata_i [25:21]* which is kept in the signal *mpc_rega_i [8:0]* after the selection of a multiplexer. The same process is implemented simultaneously in the *rt* field, and the result is kept in the signal *mpc_regb_i [8:0]*. For the *lw* instruction the *rt* field is used. Both signals are sent to the register file module *m14k_rf*. The rs field signal *mpc_rega_i [4:0]* is sent to the wire signal *src_a_reg [4:0]* after one cycle clock in the module *m14_rf*. The register file outputs the register value onto the signal *rf_adt*. Another output signal is *rf_bdt*, which is not used for this *lw* instruction, but for R-type or the *sw* instructions. The signal *mpc_ir_e [31:0]* gets the instruction data from *icc_idata_i [31:0]*. Part of the signal *mpc_ir_e [15:0]* is the immediate field of the instruction, and the required offset for the *lw* instruction is asserted to the signal *dva_offset_e*. The target register operand for the *lw* instruction is stored in *mpc_ir_e [20:16]*. For R_type instructions, the destination operand is stored in *mpc_ir_e [15:11]*. After the decode stage, the decoded instruction is executed. The signal *rf_adt* is sent to the signal *edp_abus_e*. The signal *aop_e* gets the base address from *edp_abus_e* through a multiplexer. The base address is added to the offset in *dva_offset_e* to create the virtual address signal *edp_dva_e* which is subsequently translated to the physical address from which the memory is read. In the module *m14k_mpc_ctl* the destination register signal *dest_e* gets the operand from *mpc_ir_e [20:16]*. In the memory stage, the virtual address signal *edp_dva_e* is transformed in the *m14k_mmu* module to generate the fixed physical address *mmu_dpah* which is transmitted as the signal *dcc_exaddr* through the module *m14k_dcc* to the module *m14k_biu*. In this module, the signal *daddr* gets the physical address for the *lw* instruction, and the *wtaddr* signals get the physical address for the *sw* instruction. At last, the signal *HADDR* receives the physical address for the *lw* instruction from the signal *daddr*. Through this physical address, the signal *dcc_data_m* in the module

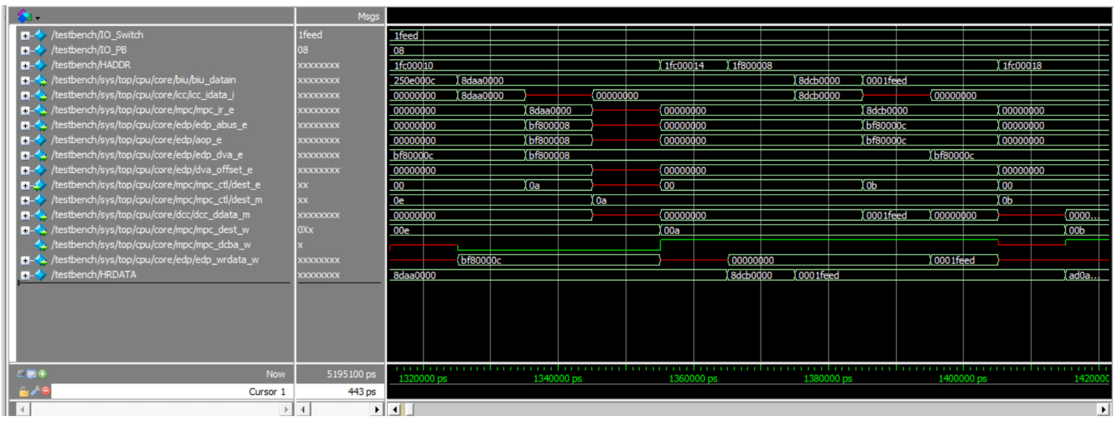

Msgs
/testbench/IO_Switch 1feed
/testbench/IO_PB 08
/testbench/HADDR xxxxxxxx
/testbench/sys/top/cpu/core/biu/biu_datain xxxxxxxx
/testbench/sys/top/cpu/core/icc/icc_idata_i xxxxxxxx
/testbench/sys/top/cpu/core/mpc/mpc_ir_e xxxxxxxx
/testbench/sys/top/cpu/core/edp/edp_abus_e xxxxxxxx
/testbench/sys/top/cpu/core/edp/aop_e xxxxxxxx
/testbench/sys/top/cpu/core/edp/edp_dva_e xxxxxxxx
/testbench/sys/top/cpu/core/edp/dva_offset_e xxxxxxxx
/testbench/sys/top/cpu/core/mpc/mpc_ctl/dest_e xx
/testbench/sys/top/cpu/core/mpc/mpc_ctl/dest_m xx
/testbench/sys/top/cpu/core/dcc/dcc_ddata_m xxxxxxxx
/testbench/sys/top/cpu/core/mpc/mpc_dest_w 0Xx
/testbench/sys/top/cpu/core/mpc/mpc_dcba_w x
/testbench/sys/top/cpu/core/edp/edp_wrdata_w xxxxxxxx
/testbench/HRDATA xxxxxxxx
Now 5195100 ps
Cursor 1 443 ps
1320000 ps
1340000 ps
1360000 ps
1380000 ps
1400000 ps
1420000

Figure 4.9: Simulation for the lw instruction

m14k_dcc receives the data from the input signals of the *mipsfpga_ahb* module and sends it to *edp_ldcpdata_w*. At the same time, the signal *dest_w* gets the destination operand from *dest_e*. At last, in the writeback stage, the signal *mpc_dcba_w* decides that *edp_lacpdata_w* is the source signal for the writeback signal *edp_wrdata_w*. The store process of the *sw* instruction is introduced in the fourth section.

4.4 Generation Process of Physical Address for the lw and sw instructions

This section describes the generation process of the physical address for the *lw* and *sw* instructions. For the *lw* instruction, this process is only used when the data is taken from the hardware input. If the data comes from the software input, it will be directly fetched at the beginning. As shown in Figure 4.10, the signal *edp_abus_e* gets the instruction from the module *mipsfpga_ahb* as described in the last section and is sent to the signal *aop_e*, which contains the base address. The signal *edp_dva_e* gets the virtual address after addition to the address offset, which is stored in *dva_offset_e*. The module *m14k_mmu* implements the transformation from the virtual address to the physical address. Table 4.3 lists the main signals that are updated during the signal conversion. The subcomponent *m14k_mmuc* and subcomponent *m14k_dtlb* are instances inside the *m14k_mmu* module. The input signal *edp_dva_e [31:0]* is sent to the signal *mmu_dva_e [31:0]*. Through an instantiation, the output signal *mmu_dva_e* from the subcomponent *m14k_mmuc* is connected with the input signal *dva_trans*. But this input signal only accepts the bit from 31 to 10, and *dva_trans [28:10]* is used during the assignment of the output signal *pre_dpah [31:10]*. The remaining bits of *pre_dpah [31:29]* are defined by the signal *dva_trans_mapped [31:29]*, which originally connected to the signal *edp_dva_mapped_e*. In the additional subcomponent *m14k_dtlb*, which is connected with the next level subcomponent *m14k_dtlb_utlb*, the output signal *mmu_dpah [31:10]* receives the translated address from *pre_dpah*, which results from an 5 inputs or-gate. The translated signal *mmu_dpah[31:10]* is transmitted to the module *m14k_dcc* and is combined with

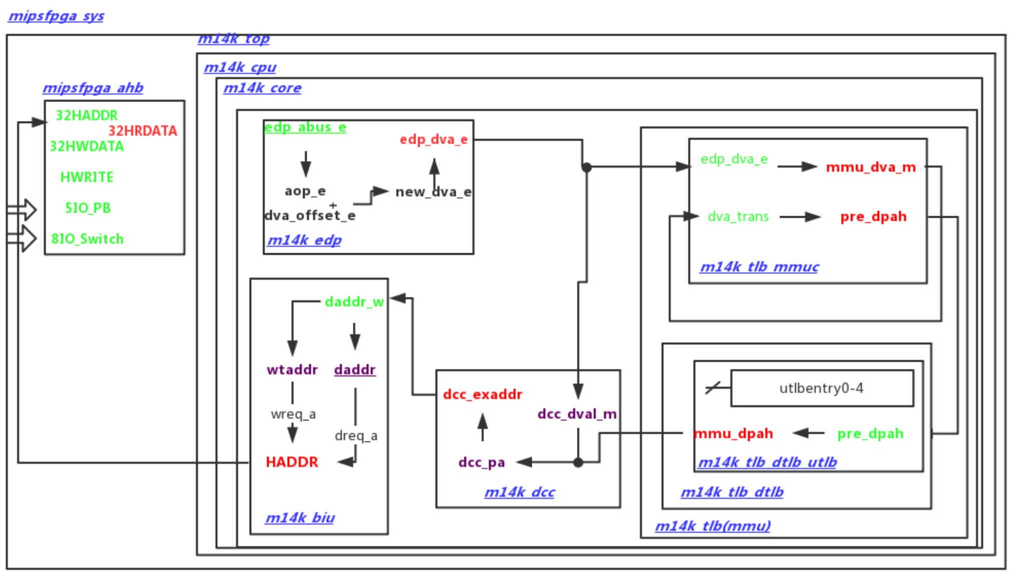
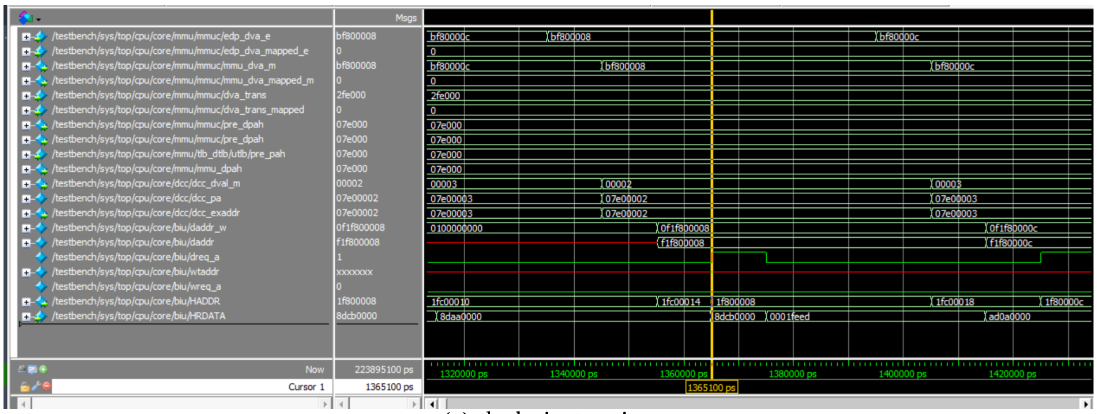


Figure 4.10: Diagram for lw and sw instructions

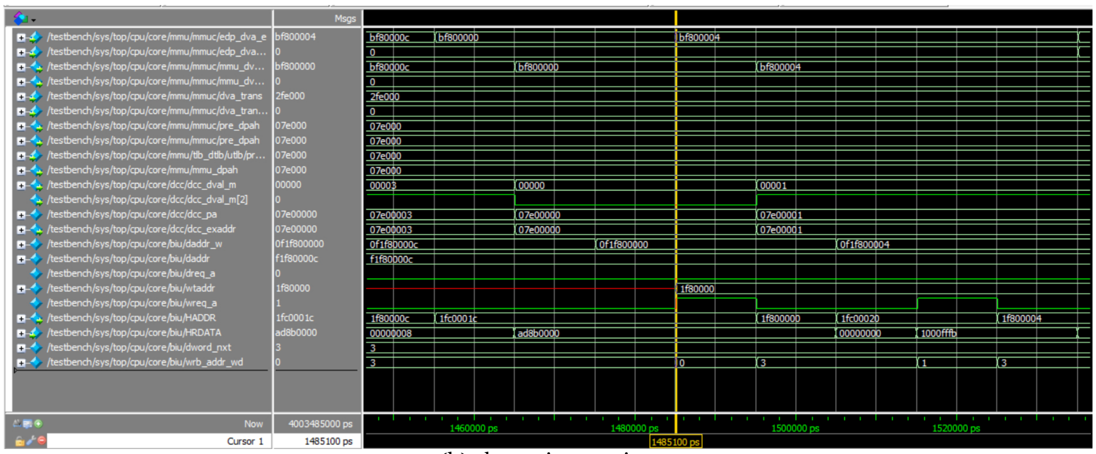
signal	Binary / Hex	31_28	27_24	23_20	19_16	15_12	11_8	7_4	3_0
edp_dva_e	bf800008	1011	1111	1000	0000	0000	0000	0000	0000
mmu_dva_e	bf800008	1011	1111	1000	0000	0000	0000	0000	0000
signal	Binary / Hex	31_30	29_26	26_22	21_18	17_14	13_10		
dva_trans	2fe000	10	1111	1110	0000	0000	0000		
dva_trans_mapped	0	00	0xxx						
pre_dpah	07e000	00	0111	1110	0000	0000	0000		
mmu_dpah	07e000	00	0111	1110	0000	0000	0000		

Table 4.4: Transform from edp_dva_e to mmu_dpah



```
                                                          Msgs
/testbench/sys/top/cpu/core/mmu/mmuc/edp_dva_e            bf800008
/testbench/sys/top/cpu/core/mmu/mmuc/edp_dva_mapped_e     0
/testbench/sys/top/cpu/core/mmu/mmuc/mmu_dva_m            bf800008
/testbench/sys/top/cpu/core/mmu/mmuc/mmu_dva_mapped_m     0
/testbench/sys/top/cpu/core/mmu/mmuc/dva_trans            2fe000
/testbench/sys/top/cpu/core/mmu/mmuc/dva_trans_mapped     0
/testbench/sys/top/cpu/core/mmu/mmuc/pre_dpah             07e000
/testbench/sys/top/cpu/core/mmu/mmuc/pre_dpah             07e000
/testbench/sys/top/cpu/core/mmu/tlb_dtlb/utlb/pre_pah     07e000
/testbench/sys/top/cpu/core/mmu/mmu_dpah                  07e000
/testbench/sys/top/cpu/core/dcc/dcc_dval_m                00002
/testbench/sys/top/cpu/core/dcc/dcc_pa                    07e00002
/testbench/sys/top/cpu/core/dcc/dcc_exaddr                07e00002
/testbench/sys/top/cpu/core/biu/daddr_w                   0f1f800008
/testbench/sys/top/cpu/core/biu/daddr                     f1f800008
/testbench/sys/top/cpu/core/biu/dreq_a                    1
/testbench/sys/top/cpu/core/biu/wtaddr                    xxxxxxx
/testbench/sys/top/cpu/core/biu/wreq_a                    0
/testbench/sys/top/cpu/core/biu/HADDR                     1f800008
/testbench/sys/top/cpu/core/biu/HRDATA                    8dcb0000

                                                   Now    223895100 ps
                                              Cursor 1    1365100 ps
```

(a) the lw instruction



```
                                                          Msgs
/testbench/sys/top/cpu/core/mmu/mmuc/edp_dva_e            bf800004
/testbench/sys/top/cpu/core/mmu/mmuc/edp_dva...           0
/testbench/sys/top/cpu/core/mmu/mmuc/mmu_dv...            bf800000
/testbench/sys/top/cpu/core/mmu/mmuc/mmu_dv...            0
/testbench/sys/top/cpu/core/mmu/mmuc/dva_trans            2fe000
/testbench/sys/top/cpu/core/mmu/mmuc/dva_tran...          0
/testbench/sys/top/cpu/core/mmu/mmuc/pre_dpah             07e000
/testbench/sys/top/cpu/core/mmu/mmuc/pre_dpah             07e000
/testbench/sys/top/cpu/core/mmu/tlb_dtlb/utlb/pr...       07e000
/testbench/sys/top/cpu/core/mmu/mmu_dpah                  07e000
/testbench/sys/top/cpu/core/dcc/dcc_dval_m                00000
/testbench/sys/top/cpu/core/dcc/dcc_dval_m[2]             0
/testbench/sys/top/cpu/core/dcc/dcc_pa                    07e00000
/testbench/sys/top/cpu/core/dcc/dcc_exaddr                07e00000
/testbench/sys/top/cpu/core/biu/daddr_w                   0f1f800000
/testbench/sys/top/cpu/core/biu/daddr                     f1f80000c
/testbench/sys/top/cpu/core/biu/dreq_a                    0
/testbench/sys/top/cpu/core/biu/wtaddr                    1f80000
/testbench/sys/top/cpu/core/biu/wreq_a                    1
/testbench/sys/top/cpu/core/biu/HADDR                     1fc0001c
/testbench/sys/top/cpu/core/biu/HRDATA                    ad8b0000
/testbench/sys/top/cpu/core/biu/dword_nxt                 3
/testbench/sys/top/cpu/core/biu/wrb_addr_wd               0

                                                   Now    4003485000 ps
                                              Cursor 1    1485100 ps
```

(b) the sw instruction

Figure 4.11: Simulation for the physical address of lw and sw instructions

the signal *dcc_dval_m [19:2]* to get *dcc_pa [31:2]*. The result is chosen by *dcc_exaddr [31:2]* through a multiplexer. In module *m14k_biu*, the input signal *dcc_exaddr [31:2]* is used as part of the wire signal *daddr_w [31:2]*. The remaining part of the wire signal is shown in Table 4.4. The virtual address for the *lw* instruction is carried by the signal *daddr[31:4]*, and for the *sw* instruction, is carried by the signal *wtaddr [31:4]*. Both signals get the address from *daddr_w [31:4]*. For both signals the bit[3:2] are defined by *dword_nxt [1:0]* and *wrb_addr_wb [1:0]* respectively. As is shown in Figure 4.11 (a)-(b). They are selected according to the signal *dreq_a* and *wreq_a* and are sent to the signal *burst_addr_nxt [31:2]*. At last, *HADDR [31:0]* gets the physical address for *lw* and *sw* instructions.

4.5 Generation Process of the Signal HWDATA and the Data for Read

One way to get the data from the periphery is the *lw* instruction. The data path is shown in Figure 4.12. At first, the module *mipsfpga_ahb* gets the physical address. The input data

signal	Binary / Hex	31_28	27_24	23_20	19_18	17_14	13_10	9_6	5_2
mmu_dpah	1f8	0001	1111	1000					
dcc_dval_m	00002				00	0000	0000	0000	0010
signal	Binary / Hex	31_30	29_26	26_22	21_18	17_14	13_10	9_6	5_2
dcc_pa	07e00002	00	0111	1110	0000	0000	0000	0000	0010
dcc_exaddr	07e00002	00	0111	1110	0000	0000	0000	0000	0010
signal	Binary / Hex	31_28	27_24	23_20	19_16	15_12	11_8	7_4	3_0
daddr_w[31:2]	1f800008	0001	1111	1000	0000	0000	0000	0000	10xx
daddr[31:4]	1f80000	0001	1111	1000	0000	0000	0000	0000	
dword_nxt	2								10xx
wtaddr	1f80000	0001	1111	1000	0000	0000	0000	0000	
dword_nxt	0								00xx
burst_addr_nxt	1f80000x	0001	1111	1000	0000	0000	0000	0000	xxxx
HADDR	1f80000x	0001	1111	1000	0000	0000	0000	0000	xxxx

Table 4.5: Transform from mmu_dpah to HADDR

signal is sent to the output signal *HRDATA* based on the address. The output signal is transmitted to the module *m14k_core*, and the data is passed from the module *m14k_biu* through *m14k_dcc* to *m14k_edp* without any change. Another way to get the data is as a constant in the software. The data is then directly written in the program and is extracted during decoding. During the execution of the *sw* instruction, the data will be written from the core to the periphery through the AHB module. As shown in Figure 4.13, the output signal *rf_bdt_e [31:0]* of the module *m14k_rf*, which carries the data, is sent to the input signal *edp_bbus_e [31:0]* of the *m14k_edp* module. This data is transmitted from the module *m14k_edp* through module *m14k_dcc* to *m14k_biu* without any change. At last, the data is sent to signal *HWDATA* and is accepted by the periphery. This process is also the same for the execution of the *sw* instruction, when the data is stored in the memory through the signal *HWDATA*. In Figure 4.14 (a) & (b) the signaling sequence which is executed in Figure 4.13 is shown.

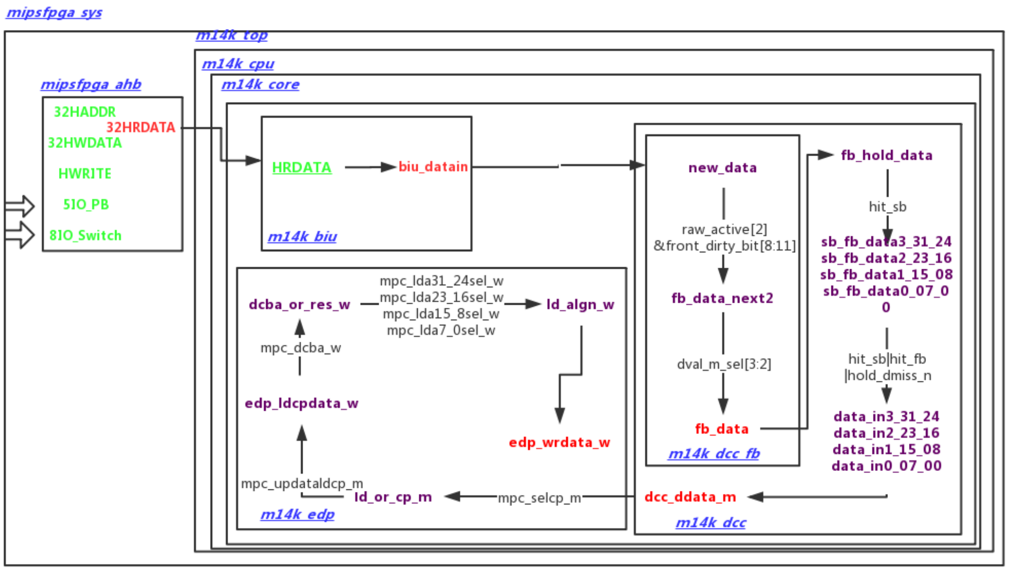


Figure 4.12: Diagram for data by using of the lw instruction

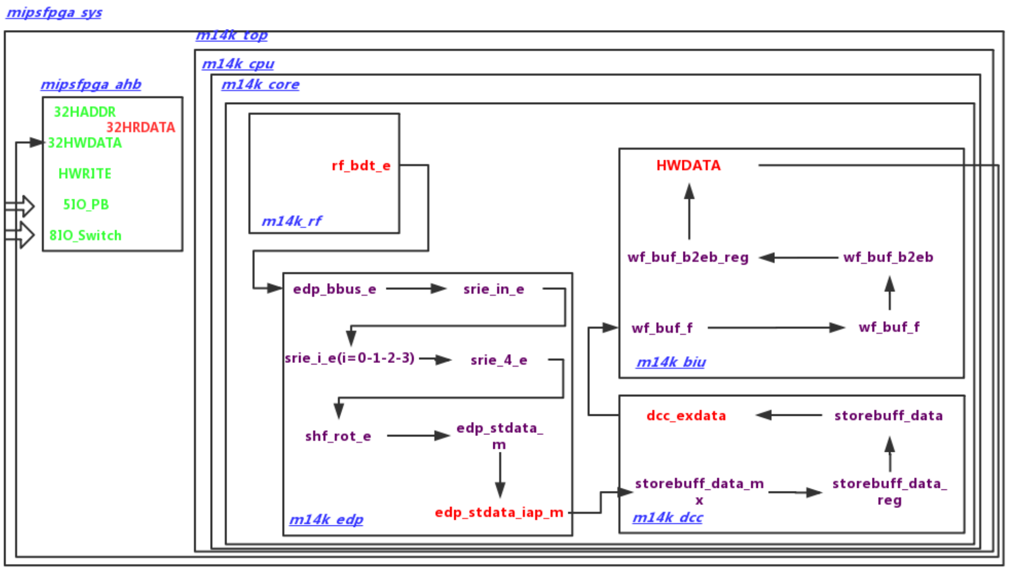
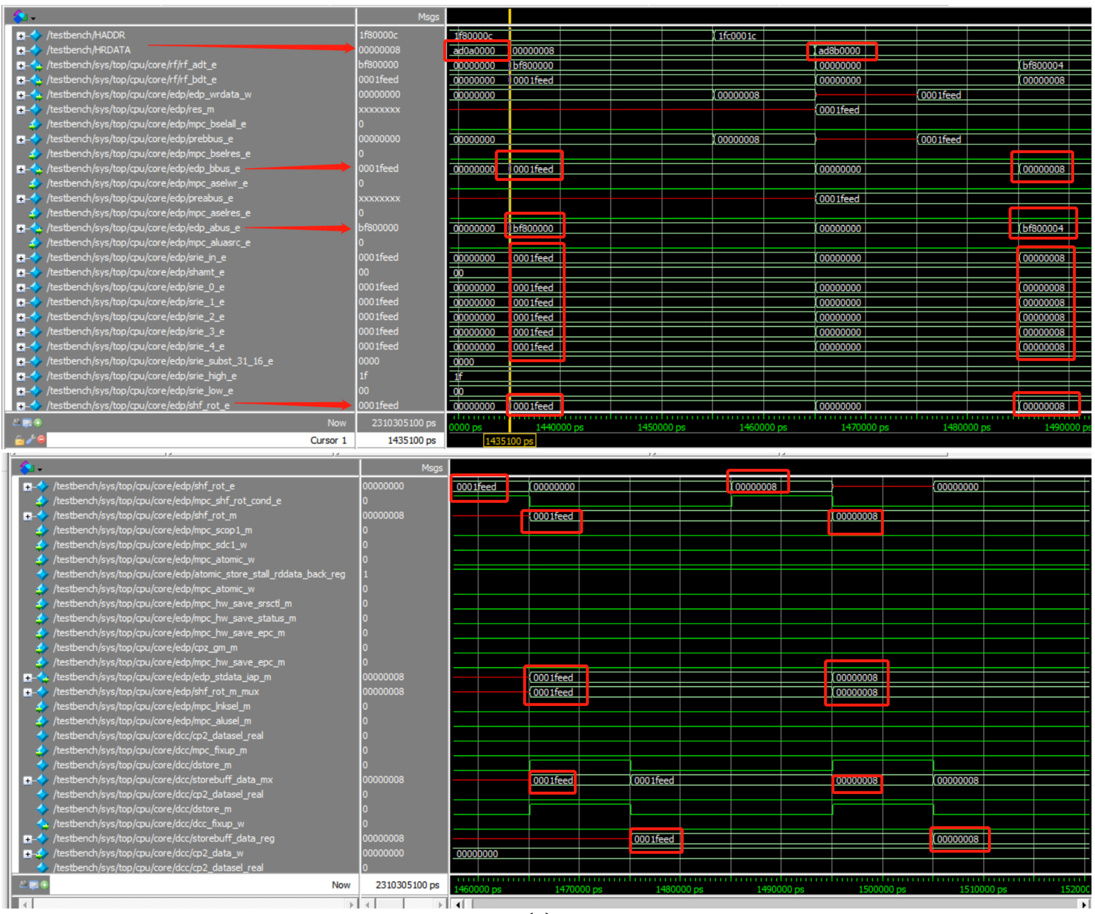


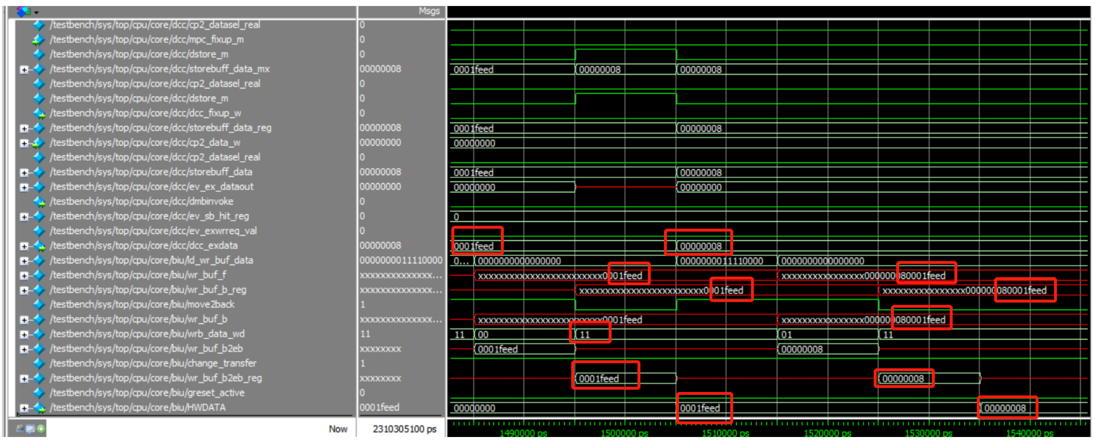
Figure 4.13: Diagram for HWDATA



	Msgs
/testbench/HADDR	1f80000c
/testbench/HRDATA	00000008
/testbench/sys/top/cpu/core/rf/rf_adt_e	bf800000
/testbench/sys/top/cpu/core/rf/rf_bdt_e	0001feed
/testbench/sys/top/cpu/core/edp/edp_wrdata_w	00000000
/testbench/sys/top/cpu/core/edp/res_m	xxxxxxxx
/testbench/sys/top/cpu/core/edp/mpc_bselall_e	0
/testbench/sys/top/cpu/core/edp/prebbus_e	00000000
/testbench/sys/top/cpu/core/edp/mpc_bselres_e	0
/testbench/sys/top/cpu/core/edp/edp_bbus_e	0001feed
/testbench/sys/top/cpu/core/edp/mpc_aselwr_e	0
/testbench/sys/top/cpu/core/edp/preabus_e	xxxxxxxx
/testbench/sys/top/cpu/core/edp/mpc_aselres_e	0
/testbench/sys/top/cpu/core/edp/edp_abus_e	bf800000
/testbench/sys/top/cpu/core/edp/mpc_aluasrc_e	0
/testbench/sys/top/cpu/core/edp/srie_in_e	0001feed
/testbench/sys/top/cpu/core/edp/shamt_e	00
/testbench/sys/top/cpu/core/edp/srie_0_e	0001feed
/testbench/sys/top/cpu/core/edp/srie_1_e	0001feed
/testbench/sys/top/cpu/core/edp/srie_2_e	0001feed
/testbench/sys/top/cpu/core/edp/srie_3_e	0001feed
/testbench/sys/top/cpu/core/edp/srie_4_e	0001feed
/testbench/sys/top/cpu/core/edp/srie_subst_31_16_e	0000
/testbench/sys/top/cpu/core/edp/srie_high_e	1f
/testbench/sys/top/cpu/core/edp/srie_low_e	00
/testbench/sys/top/cpu/core/edp/shf_rot_e	0001feed
Now	2310305100 ps
Cursor 1	1435100 ps

	Msgs
/testbench/sys/top/cpu/core/edp/shf_rot_e	00000000
/testbench/sys/top/cpu/core/edp/mpc_shf_rot_cond_e	0
/testbench/sys/top/cpu/core/edp/shf_rot_m	00000008
/testbench/sys/top/cpu/core/edp/mpc_scop1_m	0
/testbench/sys/top/cpu/core/edp/mpc_sdc1_w	0
/testbench/sys/top/cpu/core/edp/mpc_atomic_w	0
/testbench/sys/top/cpu/core/edp/atomic_store_stall_rddata_back_reg	1
/testbench/sys/top/cpu/core/edp/mpc_atomic_w	0
/testbench/sys/top/cpu/core/edp/mpc_hw_save_srsctl_m	0
/testbench/sys/top/cpu/core/edp/mpc_hw_save_status_m	0
/testbench/sys/top/cpu/core/edp/mpc_hw_save_epc_m	0
/testbench/sys/top/cpu/core/edp/cpz_gm_m	0
/testbench/sys/top/cpu/core/edp/mpc_hw_save_epc_m	0
/testbench/sys/top/cpu/core/edp/edp_stdata_iap_m	00000008
/testbench/sys/top/cpu/core/edp/shf_rot_m_mux	00000008
/testbench/sys/top/cpu/core/edp/mpc_lnksel_m	0
/testbench/sys/top/cpu/core/edp/mpc_alusel_m	0
/testbench/sys/top/cpu/core/dcc/cp2_datasel_real	0
/testbench/sys/top/cpu/core/dcc/mpc_fixup_m	0
/testbench/sys/top/cpu/core/dcc/dstore_m	0
/testbench/sys/top/cpu/core/dcc/storebuff_data_mx	00000008
/testbench/sys/top/cpu/core/dcc/cp2_datasel_real	0
/testbench/sys/top/cpu/core/dcc/dstore_m	0
/testbench/sys/top/cpu/core/dcc/dcc_fixup_w	0
/testbench/sys/top/cpu/core/dcc/storebuff_data_reg	00000008
/testbench/sys/top/cpu/core/dcc/cp2_data_w	00000000
/testbench/sys/top/cpu/core/dcc/cp2_datasel_real	0
Now	2310305100 ps

(a)



	Msgs
/testbench/sys/top/cpu/core/dcc/cp2_datasel_real	0
/testbench/sys/top/cpu/core/dcc/mpc_fixup_m	0
/testbench/sys/top/cpu/core/dcc/dstore_m	0
/testbench/sys/top/cpu/core/dcc/storebuff_data_mx	00000008
/testbench/sys/top/cpu/core/dcc/cp2_datasel_real	0
/testbench/sys/top/cpu/core/dcc/dstore_m	0
/testbench/sys/top/cpu/core/dcc/dcc_fixup_w	0
/testbench/sys/top/cpu/core/dcc/storebuff_data_reg	00000008
/testbench/sys/top/cpu/core/dcc/cp2_data_w	00000000
/testbench/sys/top/cpu/core/dcc/cp2_datasel_real	0
/testbench/sys/top/cpu/core/dcc/storebuff_data	00000008
/testbench/sys/top/cpu/core/dcc/ev_ex_dataout	00000000
/testbench/sys/top/cpu/core/dcc/dmbinvoke	0
/testbench/sys/top/cpu/core/dcc/ev_sb_hit_reg	0
/testbench/sys/top/cpu/core/dcc/ev_exwrreq_val	0
/testbench/sys/top/cpu/core/dcc/dcc_exdata	00000008
/testbench/sys/top/cpu/core/biu/ld_wr_buf_data	000000011110000
/testbench/sys/top/cpu/core/biu/wr_buf_f	xxxxxxxxxxxxxx...
/testbench/sys/top/cpu/core/biu/wr_buf_b_reg	xxxxxxxxxxxxxx...
/testbench/sys/top/cpu/core/biu/move2back	1
/testbench/sys/top/cpu/core/biu/wr_buf_b	xxxxxxxxxxxxxx...
/testbench/sys/top/cpu/core/biu/wrb_data_wd	11
/testbench/sys/top/cpu/core/biu/wr_buf_b2eb	xxxxxxxx
/testbench/sys/top/cpu/core/biu/change_transfer	1
/testbench/sys/top/cpu/core/biu/wr_buf_b2eb_reg	xxxxxxxx
/testbench/sys/top/cpu/core/biu/greset_active	0
/testbench/sys/top/cpu/core/biu/HWDATA	0001feed
Now	2310305100 ps

(b)

Figure 4.14: Simulation for HWDATA

5 Physical Implementation

The above chapters introduce the overall structure of the used MIPS core and its internal operation, This chapter presents the entire physical implementation flow in TSMC 65nm CMOS technology, based on the Cadence tool like Genus, Innovus for synthesis and place & route and Xcelium for simulation. Detailed descriptions and results are given for each implementation steps.

5.1 Memory Modules Exchange

Due to TSMC 65nm technology, provided by the manufacturer under NDA, the memories in the MIPSfpga design need to be changed from Altera's simple dual-port memory template or from the generic Xilinx RAM model to the TSMC 65nm low leakage and low voltage dual-port SRAM. There are two memory modules in the AHB Lite Bus interface which are based on Altera's memory module.

- ram_dual_port.v
- ram_reset_dual_port.v

Both memory modules RAM1 and RAM0 are mentioned in section 2.4, respectively. They use the same memory block, RAMB36E1, which is a 36Kb-bit configurable synchronous block Ram [5]. This element is used as a dual-port memory in both modules, cascaded to form a large ram block. The code setting of the memory element RAMB36E1 has an 8-bit width for data and a 6-bit width for address and a size of 64*8=512 bits. This memory block applies the read-during-write behavior [6], where the read and write addresses are the same and the output q does not get the data input d at the same clock cycle. The following listing shows the Verilog code of the module *ram_reset_dual_port.v*.

```
module ram_reset_dual_port
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=6)
(
        input       [(DATA_WIDTH-1):0] data,
        input       [(ADDR_WIDTH-1):0] read_addr, write_addr,
        input                          we, clk,
        output reg  [(DATA_WIDTH-1):0] q
);
        reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
        initial
        begin
        $readmemh("ram_reset_init.txt", ram);
        end
        always @ (posedge clk)
        begin
                // Write
                if (we)
                ram[write_addr] <= data;
                q <= ram[read_addr];
        end
endmodule
```

Listing 5: Verilog Code for RAMB36E1

Two more memory components are used in the cache, which are RAMB18E1, RAM128*1S [5] and are only used as a single dual-port memory, as shown in the following Verilog code.

```
‘timescale 1ns/10ps
module RAMB4K_S8(WE, EN, RST,
    CLK, ADDR, DI, DO);
input           WE;
input           EN;
input           RST;
input           CLK;
input  [8:0]    ADDR;
input  [7:0]    DI;
output [7:0]    DO;
reg    [7:0]    mem[0:511];
reg    [7:0]    DO;
always @(posedge CLK) begin
DO <= #1 mem[ADDR];
if (EN)
begin
if (WE)
mem[ADDR] <=#1  DI;
end
end
endmodule
```

Listing 6: RAMB4K_S8.v

```
‘timescale 1ns/10ps
module RAMB4K_S16(WE, EN, RST
    , CLK, ADDR, DI, DO);
input           WE;
input           EN;
input           RST;
input           CLK;
input  [7:0]    ADDR;
input  [15:0]   DI;
output [15:0]   DO;
reg    [15:0]   mem[0:255];
reg    [15:0]   DO;
always @(posedge CLK) begin
DO <= #1 mem[ADDR];
if (EN)
begin
if (WE)
mem[ADDR] <=#1  DI;
end
end
endmodule
```

Listing 7: RAMB4K_S16.v

Two types of TSMC memory modules can be found in Appendix A, which are tsdn65lpa65535x 16m32s with 16-bit input address and tsdn65lpll8192x16m16s with 13-bit input address. Their input and output descriptions corresponding to Port A are shown in Table 5.1.

Pin	Type	Description
AA[M-1:0]	Input	Address on Port A
DA[N-1:0]	Input	Data In on Port A
BWEBA[N-1:0]	Input	Bit Write Enable Bar on Port A(Active-Low)
WEBA	Input	Write Enable Bar on Port A (Write=0/Read=1)
CEBA	Input	Chip Enable Bar on Port A (Active-Low)
CLKA	Input	Clock on Port A
QA[N-1:0]	Output	Data Out on Port A, and D/B-WEB Compression Output for Port A

Table 5.1: TSDN65LPLLDRSRAM pin descriptions for port A [7]

M-address bit, N-data bit

Since the upper modules require at least a 15-bit address and 32-bit data, two new SRAM modules are needed, and an additional interface module shown in Appendix B is added to match the memory tsdn65lpa65536x16m32s so that it can connect to the upper-level module. The first is an additional interface module that generates signals corresponding to the input and output ports of the new SRAMs and adds two SRAM instantiations. The code shown in Appendix B corresponds to the instantiation of the memory blocks connected to the AHB for the upper-level *mipsfpga_ahb_ram_reset* module, where "*WEBA*" corresponds to the "*we*" signal of the original RAM, which is controlled by the opposite result of *HWRITE* ANDs *HSEL*. The signal *BWEBA* and *CEBA* are low active. Also, since the data input and output ports of the upper module are 32bit wide, it is necessary to divide the input *HWDATA* signal into two parts corresponding to the port width of the new SRAM, and in order to output *HRDATA*, it is necessary to add two additional wires *HRDATA_1* and *HRDATA_2* so that they can be combined and passed to the data output signal *HRDATA*, which at last is connected to the output of the new SRAM through the interface module. In addition, since the upper module address signal is only 15 bits wide, while the memory needs 16 bits, it is necessary to add a 0 to the highest bit for the complete address definition. In addition port B of the memory is not used. However, it is essential to set a constant value for all input signals of port B to prevent it from receiving undefined values, and clock port *CLKB* also needs to be connected to the system clock to avoid timing violations. The replacement of the memory in the cache will be simpler in comparison to the AHB module because the upper-level modules required data bus width are smaller than those in the memory module and do not need an additional interface module. The memory block can be directly instantiated in the upper level module. The code in Appendix B shows one of the memory module instances in the upper-level modules.

5.2 Functional Simulation

To test the logic unit functionality of the whole design without considering the time constraints, functional simulation by means of an EDA tool like Cadence Xcelium needs to be completed. Afterwards the design implementation including synthesis and P&R process can be started. This section presents the results of two simulations, which are also applied on the gate-level netlist after synthesis and in the power consumption extraction after place & route.

5.2.1 Results of functional simulation

- **Functional Simulation for the lw and sw instructions**
  First, since the current memory can only read 16-bit addresses and data while the initial file mentioned in section 4.2.1 uses words of 32-bit, the initial files have to be split into two parts, which contain a high and a low 16-bit halfword. Then the function "*load*" is used in the testbench to load the file into the memory model as shown in Appendix C. Moreover, the memories to be initialized at a specific time of 455ns due to the memory reset which happens earlier.

```
initial
begin
#455
testbench_read.mipsfpganexys4_read.mipsfpga_sys.mipsfpga_ahb.
   mipsfpga_ahb_ram_reset.sram_interface_15.
   TSDN65LPA65536X16M32S_1.MX.load("../rtl_up/initfiles/3
   _Switches&LEDs/ram_reset_init_read_low.txt");
end
```

Listing 8: Code for load initial file in the testbench

At last, the paths to the testbench, to all module files, and to the memory module files are defined in the execution script file "*make*" as shown in Appendix D, while the header files are also added through the option "*-incdir*", which includes the definition of variables used in the Verilog files. As mentioned above, it is necessary to additionally split this machine code into two parts, which is done by means of the method mentioned in section 4.1. The Listing below shows the low part of the initial file loaded into SRAM. Another part can be found in chapter 4.

```
bf80
0004
0008
000c
0000
0000
0000
0000
fffb
```

Listing 9: ram_reset_init_read_low.txt

After analyzing the simulation result, it is clear that the contents of the initial file can be found in the memory connected to the AHB. As mentioned above, two memories are required, and the initial file load starts after 455ns, as shown in Figure 5.1. The values of the preset input signals switch, and pushbutton can be passed to the output read signal *HRDATA*, as shown in Figure 5.2.

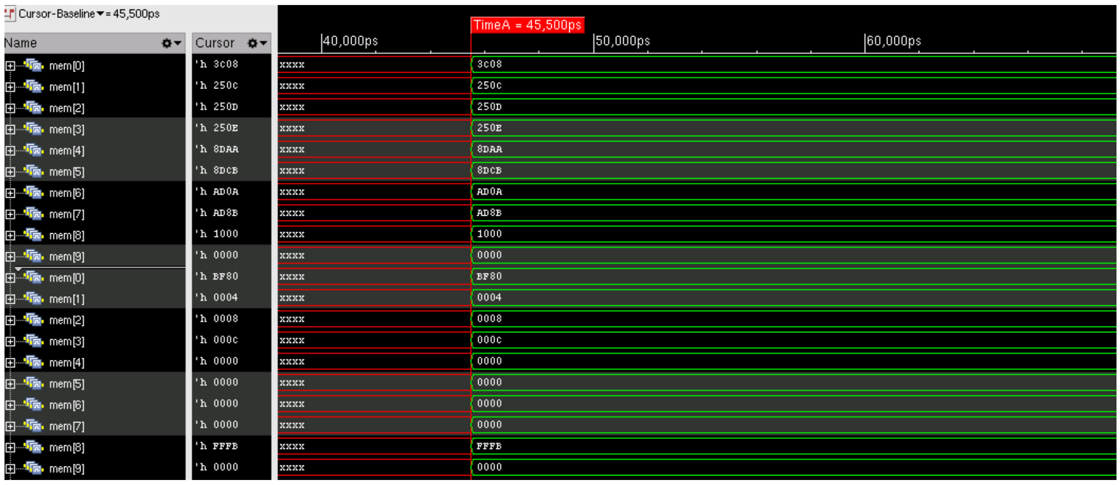


Figure 5.1: Simulation result in memory

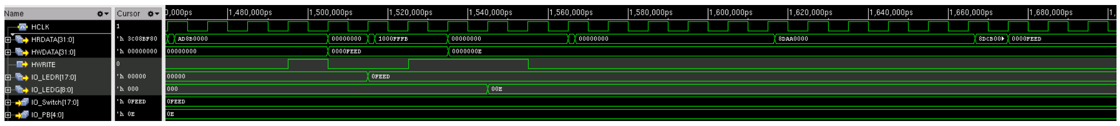
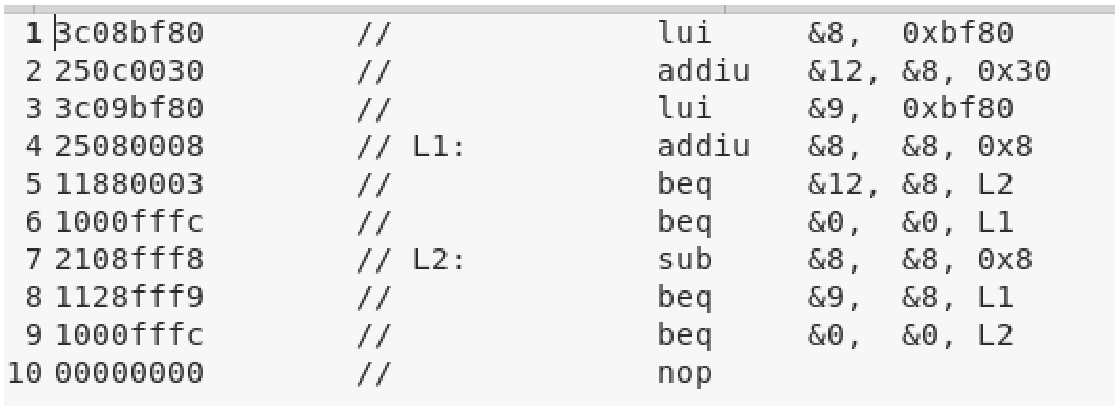

Name
Cursor
HCLK 1
HRDATA[31:0] 'h 3c08BF80 AD8B0000 00000000 1000FFFB 00000000
HWDATA[31:0] 'h 00000000 00000000 0000FEED 0000000E
HWRITE 0
IO_LEDR[17:0] 'h 00000 00000 0FEED
IO_LEDG[8:0] 'h 000 000 00E
IO_Switch[17:0] 'h 0FEED 0FEED
IO_PB[4:0] 'h 0E 0E

Figure 5.2: Simulation result for input and output

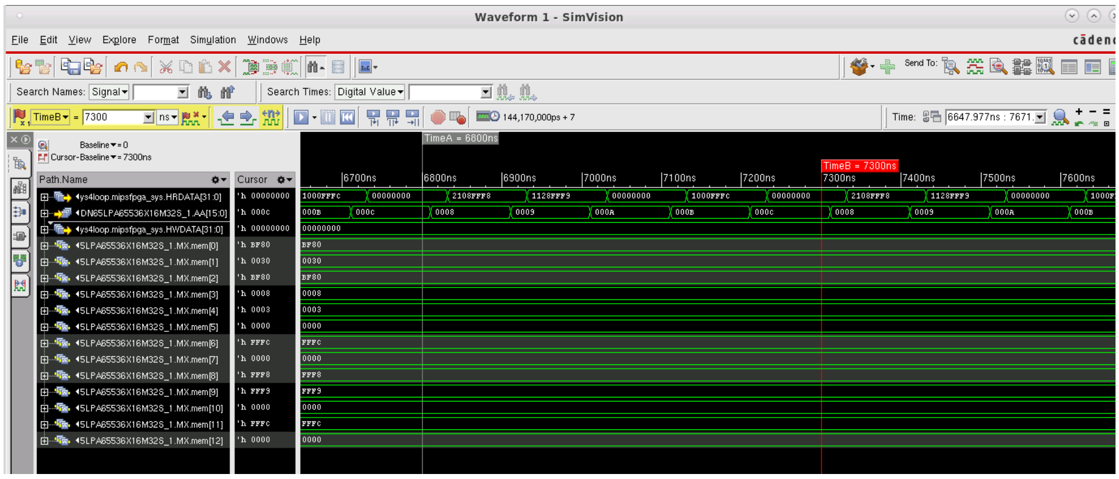
- **Functional Simulations for the Register-Centric Program**
  This assembly program aims to execute instructions on data stored in registers without any memory access in an infinite loop. At the label L1 of the program an address is incremented from *0xbf800000* to *0xbf800030* and compared in the next line of the program to check if the address is the same in both registers, and if they are the same, then the program jumps to label L2, while at the label L2 the address is decremented and afterwards a jump to the start of the loop is executed. The left half of Figure 5.3 shows the machine code obtained by generating the initial file described in subsection 4.1. Due to the different address bit sizes, the machine code also needs to be split into two parts before the simulation. The lower 16-bit half-words are loaded into the SRAM and are shown in Figure 5.4. Thus, by comparing the read data bus signal *HRDATA* and the input address signal *AA* of the SRAM, it is clearly seen that at 6800 ns, the machine code 2108FFF8 received from the output of the SRAM is passed to *HRDATA*, when the

address 0008 at the signal *AA* is received, other addresses can also correspond to the code. Due to the looping process, a new cycle starts at 7300 ns.



```
3c08bf80      //           lui     &8,  0xbf80
250c0030      //           addiu   &12, &8, 0x30
3c09bf80      //           lui     &9,  0xbf80
25080008      // L1:       addiu   &8,  &8, 0x8
11880003      //           beq     &12, &8, L2
1000fffc      //           beq     &0,  &0, L1
2108fff8      // L2:       sub     &8,  &8, 0x8
1128fff9      //           beq     &9,  &8, L1
1000fffc      //           beq     &0,  &0, L2
00000000      //           nop
```

Figure 5.3: Assembly Program for Address Cycling



Path.Name	Cursor
ys4loop.mipsfpga_sys.HRDATA[31:0]	'h 00000000
DN65LPA65536X16M32S_1.AA[15:0]	'h 000c
ys4loop.mipsfpga_sys.HWDATA[31:0]	'h 00000000
5LPA65536X16M32S_1.MX.mem[0]	'h BF80
5LPA65536X16M32S_1.MX.mem[1]	'h 0030
5LPA65536X16M32S_1.MX.mem[2]	'h BF80
5LPA65536X16M32S_1.MX.mem[3]	'h 0008
5LPA65536X16M32S_1.MX.mem[4]	'h 0003
5LPA65536X16M32S_1.MX.mem[5]	'h 0000
5LPA65536X16M32S_1.MX.mem[6]	'h FFFC
5LPA65536X16M32S_1.MX.mem[7]	'h 0000
5LPA65536X16M32S_1.MX.mem[8]	'h FFF8
5LPA65536X16M32S_1.MX.mem[9]	'h FFF9
5LPA65536X16M32S_1.MX.mem[10]	'h 0000
5LPA65536X16M32S_1.MX.mem[11]	'h FFFC
5LPA65536X16M32S_1.MX.mem[12]	'h 0000

Figure 5.4: Functional Simulation Result for address cycling

5.3 Synthesis

After the correct completion of the verification based on the functional simulations, this chapter focuses on the logic synthesis process, which is the process of mapping the RTL code through EDA tools like Cadence Genus to a gate-level circuit composed of logic gates based on the TSMC standard cell libraries so that the generated results can be used in the implementation process and also for other verification processes. In order to reduce dynamic power, the clock propagation is reduced by using the clock gating technique for flip-flops that are not updated. The following three steps are the main steps of synthesis.

- **Transition**
  HDL code conversion to generic Boolean gate arrays
- **Optimization**
  Optimize the circuit and reduce unnecessary parts
- **Mapping**
  Mapping Boolean equations to logic gates according to constraints and technical libraries

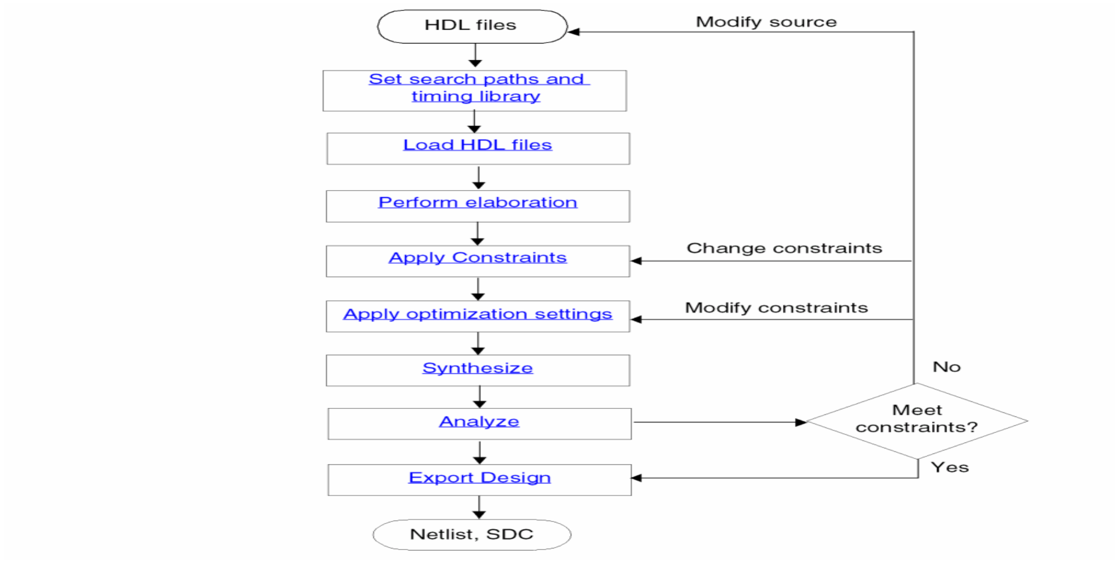


Figure 5.5: Synthesis flow in Genus [8]

As shown in Figure 5.5, the path to the required library files, like LEF, lib. files, all Verilog files, and the memory module files need to be added and set in the setup script file before the synthesis process can be performed, and finally, the netlist is exported as a Verilog file and also constraints are automatically generated for the next implementation step. The function, timing and power characteristics of logic gates are defined in Liberty libraries while the physical representation of the cell is defined in LEF (Library Exchange Format) format [9]. The Listing below shows the basic setting for the files required during the synthesis.

```
set_db / .hdl_language v2001
set_db / .init_lib_search_path ".$DIGLIBS␣$MACROLIBS"
set_db / .script_search_path ".${synpath}/scripts"
set_db / .init_hdl_search_path ".$RTL_PATH␣$MACRO_PATH"
#### Logging information level - suggested 7
set_db / .information_level $INFO_LEVEL
#### Reads the libraries.
#read_libs -max_libs $DIGLIBS$b$MACROLIBS
```

```
read_libs -max_libs $DIGLIBS
## PLE
set_db / .library "$DIGLIBS␣$MACROLIBS"
set_db / .lef_library $LEFLIBS
set_db / .cap_table_file $CAPTABLE
```

Listing 10: Basic files required during synthesis

The use of clock gating technique can be accomplished by setting the following option. And the variable CLOCK_GATING is defined as true.

```
set_db / .lp_insert_clock_gating $CLOCK_GATING
```

5.3.1 Timing Constraint

To meet the hold and setup time requirements of the internal unit, the clock, input, and output ports need to be constraint in the SDC file [10]. Because of delays the transmission of signals cannot be fully synchronized to each register in reality, so additional constraints need to be set to reduce the impact caused by delays or other phenomena. The following will introduce some of the commands used in this SDC file and some basic concepts of time constraints. The frequency of the clock needs to be determined at first. In this adaptation 50MHz will be used, which corresponds to a period of 20ns, meanwhile the rising and falling edges are defined as 0ns and 10ns respectively for 50% duty cycle. The corresponding command is shown below.

```
create_clock -period 20.00 -name clk -waveform {0.00 10.00}
                [get_ports clk]
```

Listing 11: Command create_clock

The next consideration is the time it takes for a signal to go from one state to another, for example, from 0 to 1, which is called slew rate, and can be set by the instruction below, for signal rise and fall, respectively. In contrast to *set_clock_transition*, which provides specification for the entire clock network, the instruction defines the transition of a specific input signal.

```
set_clock_transition 1.65 -rise [get_clocks clk]
set_clock_transition 1.69 -fall [get_clocks clk]
```

Listing 12: Command set_clock_transition

Another type of delay, known as clock latency, is caused by capacitive load of different elements. There are two types of clock latency, network latency and source latency. Network latency is the delay from the clock definition point (*create_clock*) to the flip-flop clock pin. Source latency is the delay from the clock source to the clock definition point. source latency may represent on-chip or off-chip latency. The total clock delay on the flip-flop clock pins is the sum of the source and the network delays [11]. The following command shows two types, if no source is marked then the network type is indicated.

```
set_clock_latency -source -early 0.576 [get_clocks clk]
set_clock_latency -source -late 0.029 [get_clocks clk]
```

Listing 13: Command set_clock_latency

Before explaining the other commands, two concepts need to be introduced. One is jitter, which is the difference between two clock cycles. This error is generated inside the clock generator and is related to the internal circuitry of the crystal oscillator or the PLL, while the routing has no effect on it. Another is skew, which is the difference in delay between multiple sub-clock signals drived by the same clock. Because of skew or jitter issues, the design can have setup and hold violations [11]. Due to the influence of these two factors, it is difficult to predict the exact edge time which triggers the flip-flop, a phenomenon which is known as uncertainty. Since the uncertainty is set for a single clock, setup is affected by both jitter and skew, while hold is only affected by skew, so setup settings should be chosen greater than hold settings.

```
set_clock_uncertainty -setup 0.19 [get_clocks clk]
set_clock_uncertainty -hold 0.15 [get_clocks clk]
```

Listing 14: Command set_clock_uncertainty

The concepts of setup and hold time need to be explained before introducing the input and output delay. The time is the minimum time that the signal at the data input must remain stable before the edge of the clock is valid. Hold time is the minimum time that the signal at the data input must remain stable after the valid edge of the clock. The max and min options are used in the following input, and output delay commands for the setup check and hold check, respectively, where the input delay represents the arrival time for a given port at a given clock, and the output delay is the amount of requested time before a clock edge.

```
set_input_delay -clock [get_clocks clk] -min -add_delay 3.200
                [get_ports {sw[0]}]
set_input_delay -clock [get_clocks clk] -max -add_delay 7.200
                [get_ports {sw[0]}]
```

Listing 15: Command set_input_delay

```
set_output_delay -clock [get_clocks clk] -min -add_delay 3.000
                 [get_ports {led[0]}]
set_output_delay -clock [get_clocks clk] -max -add_delay 6.600
                 [get_ports {led[0]}]
```

Listing 16: Command set_output_delay

5.3.2 Results of Synthesis

As mentioned at the beginning of this section in the first of the three main paths of the synthesis procedures, the synthesis tool converts the HDL language into generic logic cells, like AND and OR gates, or components with storage function, such as flip-flops, and then through optimization and mapping to the standard cells of the target technology, the synthesis process is completed. The optimization process also affects the hierarchy and as a result with this design, there are only 7 modules left after synthesis.

- **A Result Example**
  In Figure 5.6, a synthesized circuit built from cells from taken the standard cell library is shown. The DFQD1 component is a flip-flop clocked by the system clock *SI_ClkIn* and fed by the input signal *mpc_evec_sel [3]*. However, according to the corresponding

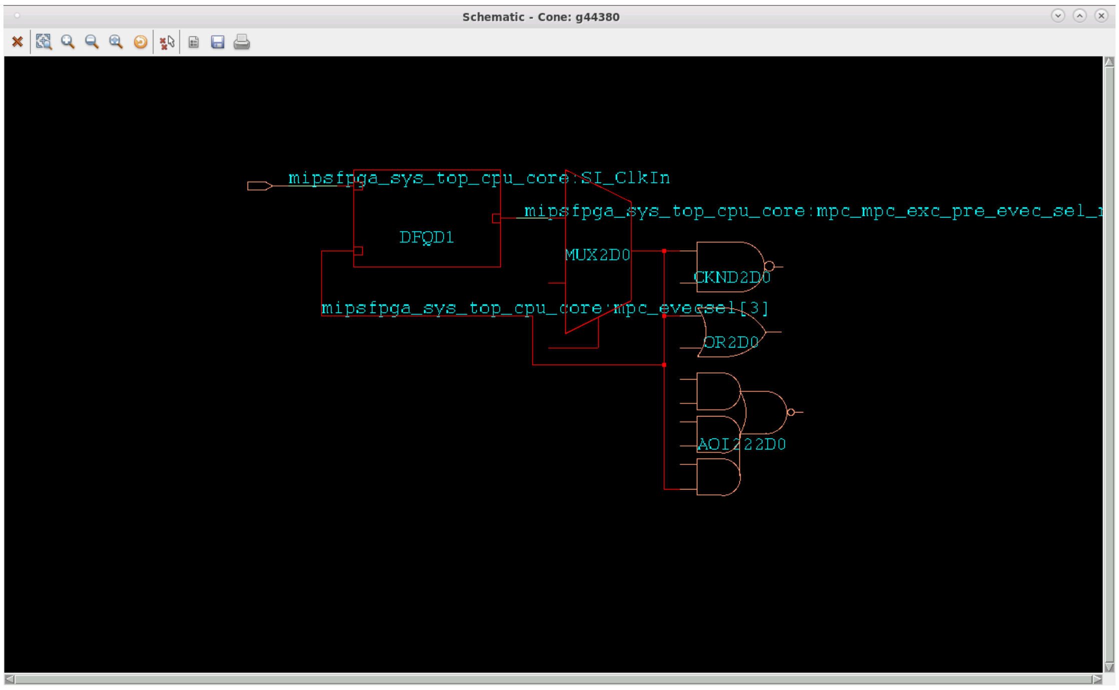

Schematic - Cone: g44380
mipsfpga_sys_top_cpu_core:SI_ClkIn
mipsfpga_sys_top_cpu_core:mpc_mpc_exc_pre_evec_sel_
DFQD1
MUX2D0
CKND2D0
mipsfpga_sys_top_cpu_core:mpc_evecsel[3]
OR2D0
AOI222D0

Figure 5.6: Synthesis schematic mpc_evec_sel[3]

Verilog code shown in Listing 17 the actual input signal should be *pre_evec_sel [3]*.

```
mvp_ucregister_wide #(.WIDTH(5)) _pre_evec_sel_reg_4_0_(.q(
    pre_evec_sel_reg[4:0]),.scanenable(gscanenable), .cond(
    new_exc_chain_id), .clk(gclk), .d(pre_evec_sel));
```

Listing 17: Verilog description for pre_evec_sel_reg

The reason for the difference is that the signal *pre_evec_sel[3]* is driving the signal *mpc_evecsel[3]*, and thus the signal is replaced during synthesis. The statement relevant for the optimization is as follows.

```
assign mpc_evecsel [7:0] = {debug_rdvec, debug_vect,
    pre_evec_sel};
```

Listing 18: Verilog description for the signal mpc_evecsel

In addition a multiplexer is also involved in the generation of the signal *pre_evec_sel[3]*, the related statement is given below:

```
assign pre_evec_sel [4:0] = new_exc_chain_id ? {au_reset_reg,
    cpz_bev, tlb_refill_vec_n, cacheerr_vect, int_vect} :
                        pre_evec_sel_reg;}
```

Listing 19: Verilog description for the signal pre_evec_sel

Thus the corresponding netlist description shown below is generated after optimization:

```
 MUX2D0 g44380(.I0 (mpc_mpc_exc_pre_evec_sel_reg[3]), .I1 (
    cpz_bev), .S (mpc_mpc_exc_new_exc_chain_id), .Z (
    mpc_evecsel[3]));
```

Listing 20: New Verilog description for the signal mpc_evecsel

- **Result of Timing Constraints**
  The following listing shows that after the execution of the command read_sdc the timing constraints are all successfully executed.

```
Statistics for commands executed by read_sdc:
 "create_clock"          - successful      1 , failed      0 (
    runtime  0.00)
 "get_clocks"            - successful    127 , failed      0 (
    runtime  0.00)
 "get_ports"             - successful    179 , failed      0 (
    runtime  0.00)
 "set_clock_latency"     - successful      3 , failed      0 (
    runtime  0.00)
 "set_clock_transition" - successful      2 , failed      0 (
    runtime  0.00)
 "set_clock_uncertainty"- successful      2 , failed      0 (
    runtime  0.00)
```

```
"set_input_delay"       - successful      54 , failed       0 (
   runtime  0.00)
"set_input_transition" - successful      27 , failed       0 (
   runtime  0.00)
"set_load"              - successful      32 , failed       0 (
   runtime  0.00)
"set_output_delay"      - successful      64 , failed       0 (
   runtime  0.00)
```

Listing 21: Result of timing constraints

5.4 Implementation (Floorplan and P&R Process)

This section will explain the process of implementation and its results, which includes two parts: floorplan and place & route (P&R). The main consideration of floorplaning is the arrangement of the I/O pads, the size and shape of the core, and the position of macro cells and blockages, and the distribution of the power and ground network. The main purpose of the power and ground network is to distribute the supply voltage from the pads to the entire chip by means of a grid which consists of metal rings around the core and sufficiently arranged horizontal and vertical stripes. The purpose of the place step is to place standard cells within the core based on the premise of the floorplan settings, and to optimize the location of each component, add necessary buffers, and also remove some unnecessary components. The route step involves establishing the connection between each component, optimizing the timing and other items, and checking whether there is a congestion problem through DRC. If so, it needs to be considered and fixed by adding stripes or blockages. Finally, necessary files for the power extraction are generated including the synthesized netlist used in the gate-level-simulation(GLS), the physical netlist, the DEF file and the extraction of power parasitics.

5.4.1 Floorplaning Results

Some basic parameters need to be set before defining the results of floorplan.

- **MMMC**
  MMMC stands for Multi Mode Multi Corner and is a flow option which assures that the design is functional under multiple operation conditions(corner) and multiple operation modes(mode). The main purpose of the MMMC file is the definition of operation conditions used in timing analysis for the worst, best, typical process and capacitor and resistance corners.
- **VDD/VSS connection**
  The following command is required to define the ground/power connection before the power planning and after design import. This definition is necessary because the initial

HDL description and usually also the synthesized netlist does not contain any information about the power connectivity. In addition the ground signal GND of the TSMC memory module tsdn65lpa65536x16m32s has also to be connected to the ground network. Tielo and Tiehi cells are used for the implementation of constant values like 0/1. These cells have also to be connected to power and ground nets.

```
connect_global_net VDD -type pg_pin -pin_base_name VDD
   -inst_base_name *
connect_global_net  VSS -type pg_pin -pin_base_name GND
   -inst_base_name *

connect_global_net  VSS -type pg_pin -pin_base_name VSS
   -inst_base_name *

connect_global_net VDD -type tiehi
connect_global_net VSS -type tielo
```

Listing 22: Command connect_global_net

- **TSMC Memory Module Placement**
  31 memory cells are placed in the core area by means of the following command which is added to the script file controlling the implementation flow. Here the option "-*place*" is for defining the object name. It should be noted that the instance names are changed during the synthesis process. The names of the hierarchical modules which are flattened during synthesis become part of the instance in addition the dot indicating the hierarchy level is exchanged by an underscore. The option "*-ref*" indicates the object to be referenced for the definition of the relative SRAM component position. Here the "*die*" edge is used as reference. Options like "*-horizontal_edge_separate*" and "*-vertical_edge_separate*" define the vertical and horizontal space adjustment respectively. To avoid wiring conflicts in P&R, and to reduce the length of the wires, it is necessary to place the memory modules such that the pin ports are facing the core area.

```
create_relative_floorplan
-place mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_
sram_interface_15_TSDN65LPA65536X16M32S_2
-ref_type die_boundary
-horizontal_edge_separate {1 -120 1}
-vertical_edge_separate {2 -120 2}
```

Listing 23: Command create_relative_floorplan

- **Add_stripes**
  To reduce the impedance of the power supply network stripes are added by the follow-

ing command:

```
add_stripes -direction vertical -number_of_sets 13 -spacing 2
    -layer M9 -width 10 -nets { VSS  VDD } -start_offset 40
    -stop_offset 40
```

Listing 24: Command add_stripes

Figure 5.7 shows the layout result of the finalized floorplan in physical mode. The width of the *die* is chosen to be 4755 µm, the height is 3290 µm, and the distance from each edge of the I/O boundary to the core area is 100 µm. In Figure 5.7, the *Halo* around the memory blocks can be seen, which is set to prevent standard cells from being placed inside the area marked by the *Halo* to reduce congestion.

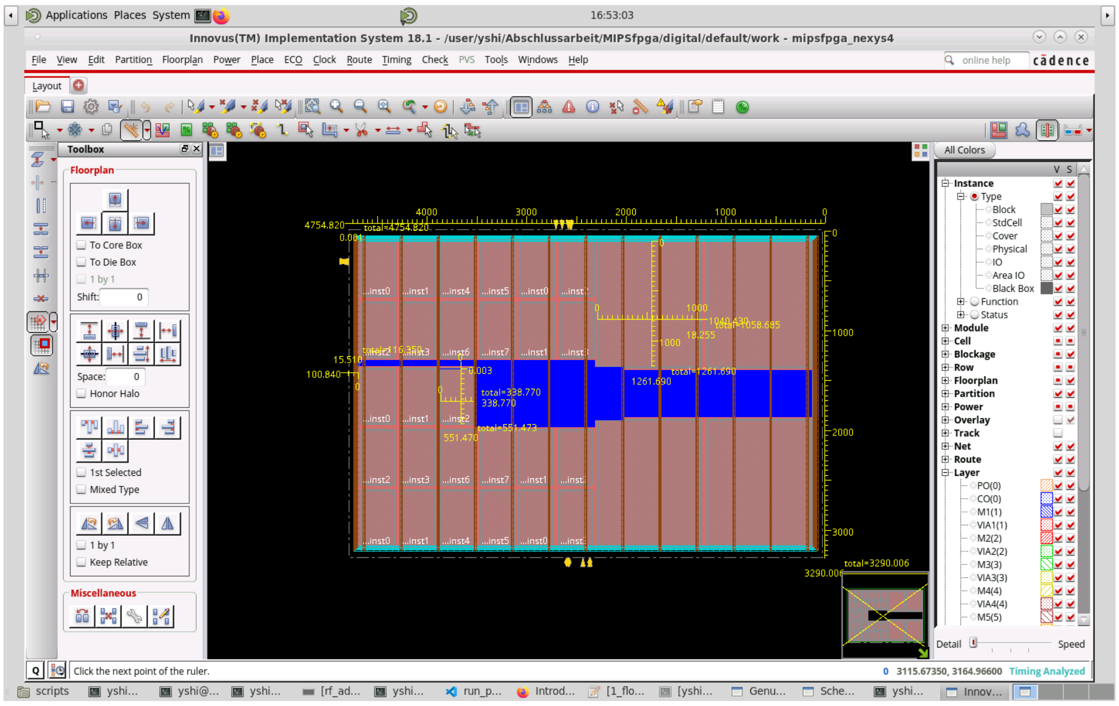


Figure 5.7: Result of floorplan

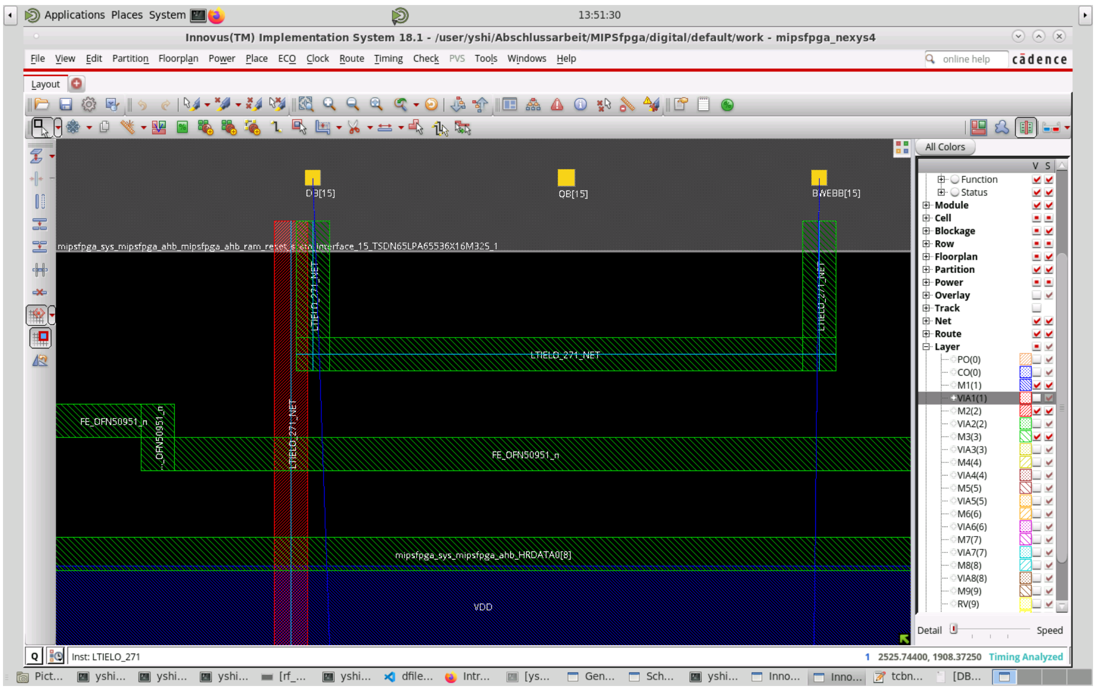
5.4.2 Results of P&R

After the place and route procedure is completed, each pin and port is connected either to each other or to VDD and VSS through different metal layers according to the location information provided in the LEF.

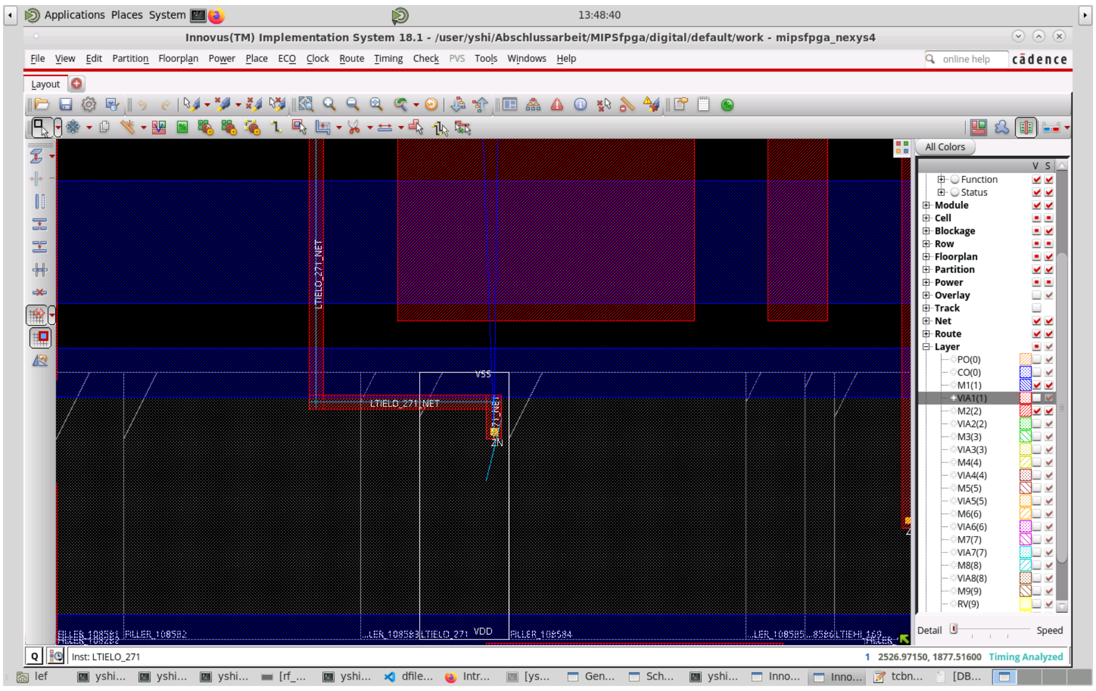
- **P&R Result of connection with VDD/VSS**
  For example, the input signals DB and BWEBB of the sram_interface module are connected to constant zero, so they are connected to VSS through TIELO cells during place and route. Figure 5.8 (a) shows that two ports are connected to each other through metal three marked in green, and Figure (a) and (b) show that both are connected to TIELO cells through metal2, which finally established the ground connection.
- **Total Negative Slack and Worst Negative Slack**
  As shown in the following table, the worst negative slack has a positive value, which means that the design passes timing checks. The total negative slack has a value of zero, which means that the design matches the timing constraints.

Hold mode	all	reg2reg	default
WNS(ns)	0.0239	0.0239	2.478
TNS(ns)	0.000	0.000	0.000
Violation Paths	0	0	0
All Paths	9698	9400	49

Table 5.2: WNS and TNS for hold mode


Innovus(TM) Implementation System 18.1 - /user/yshi/Abschlussarbeit/MIPSfpga/digital/default/work - mipsfpga_nexys4
DB[15]
QB[15]
BWEBB[15]
mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_s_sram_interface_15_TSDN65LPA65536X16M32S_1
LTIELO_271_NET
FE_OFN50951_n
mipsfpga_sys_mipsfpga_ahb_HRDATA0[8]
VDD
Inst: LTIELO_271
2525.74400, 1908.37250
Timing Analyzed

(a)


Innovus(TM) Implementation System 18.1 - /user/yshi/Abschlussarbeit/MIPSfpga/digital/default/work - mipsfpga_nexys4
LTIELO_271_NET
VSS
ZN
VDD
FILLER_108582
LTIELO_271
FILLER_108584
Inst: LTIELO_271
2526.97150, 1877.51600
Timing Analyzed

(b)

Figure 5.8: Connection between port DB[15] and BWEBB[15] of the sram_interface with VSS

6 SIMULATION

In this section, the simulation of the design after synthesis and P&R is discussed, which can also be called post-p&r-simulation, In this approach a gate-level-simulation is performed which considers the real delay introduced by the cells as well as the wiring. The X-propagation is also discussed and a solution for the problem is presented. After this process, the power consumption using Voltus will be extracted and the SRAM performance will be compared for two different instruction types.

6.1 SIMULATION SETUP

Before starting the following two simulations, it is necessary to set up basic preferences for the simulation environment. Solutions to unique problems, such as storage elements not resetting themselves and x-propagation, are described in subsequent sections.

- **Gate Level Simulation**
  The complete contents of command "*run*" below for GLS can be found in Appendix D.

```
xrun ... +rwc ../output/export.v
         /eda/kits/TSMC/.../verilog/tcbn65lp_200a/tcbn65lp.v
         ../testbench_read.v
         ../memory.v
         ...-sdf_cmd_file ../sdf_typ.cmd
         -input   ../export_saif_typ.tcl
```

Listing 25: xrun command for gate-level simulation

(1) Standard Delay Format (SDF) File
Timing and especially the propagation delay of the cells and the wiring can either be set to zero, or to a unit delay or defined precisely during the implementation procedure. For detailed timing the SDF file has to be read during elaboration, which contains the actual delay simulation information extracted during place & route which is back-annotated to the corresponding instance and net of the netlist. The procedure of reading in the timing information can be divided into two steps. The first step is to use the *xrun* command with the option "*-sdf_cmd_file*" to read the SDF file, as shown in Listing 26, the other step is applied in the Verilog file which contains the netlist by using the command "*$sdf_annotate*" to read the SDF information. The following introduces the setting for the cmd file.

```
SDF_FILE = "../design_export.sdf",
SCOPE = "testbench_read.mipsfpganexys4_read",
LOG_FILE = "sdf_annotation_typ.log",
MTM_CONTROL = "TYPICAL";
```

Listing 26: sdf_typ.cmd

The SDF file is specified by above code. In addition the scope of the simulation has to be defined to adopt to any additional hierarchy level introduced by the simulation testbench. Furthermore the log file name and path, as well as delay mode is set.
During simulation data has to be generated for use in the power extraction. For this purpose an ASCII file in SAIF (Switching Activity Interchange Format) format, which contains the switching activity information, and the .trn extension file which contains complete signals waveforms in SHM format [12] are generated by the following Tcl code. For proper annotation during power extraction the correct simulation scope has to be chosen. And since the simulation does not stop automatically, the appropriate run time needs to be added after the command run.

```
dumpsaif -scope testbench_read.mipsfpganexys4_read
         -internal -memories -overwrite -inctoggle
         -output ..sim/output/saif_db_typ.saif
database -open shm_db -into ..sim/output/shm_db_typ.cmd
         -shm
probe    -create testbench_read.mipsfpganexys4_read
         -depth all -shm -all
         -database shm_db
         -memories -unpacked 65536
run      200000ns
exit
```

Listing 27: export_saif_typ.tcl

(2) Gate level netlist, Basic library cells
The gate-level-netlist is a netlist file generated during P&R, which contains the optimized design and in turn has many differences in the instances naming e.g. due to removed hierarchy levels. The netlist is used as the basis of power extraction in combination with the SDF file which is used to back annotate timing information. The standard cell library contains gates provided by the technology vendor. The library also includes logic cells, registers, latches, flip-flops, etc, and includes functional models of each gate prepared for timing back-annotation.

(3) Testbench
The testbench is used during post-implementation timing simulation. Due to the modified instance names, some instance names e.g. of the SRAM blocks in the testbench have to be adapted in comparison to the functional simulation, as shown in the following code. As already mentioned above removed hierarchy lev-

els are introduced in the instance name and are separated by an underscore. The lowest hierarchy level of the SRAM instance is called MX and is introduced by the memory simulation model.

```
initial
begin

testbench_read.mipsfpganexys4_read.
mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_
sram_interface_15_TSDN65LPA65536X16M32S_1
.MX.load("../rtl_up/initfiles/3_Switches&LEDs
/ram_reset_init_low.txt");
....
end
```

Listing 28: Code for load in testbench during GLS

- **Script for Power Extraction**
  The vector based engine of Cadence Voltus is used for power extraction. This tool is controlled by a script which is partly explained in the following section.

```
read_mmmc ../power/scripts/mmmc9t_cui_voltus.view
read_physical -lefs {
/eda/kits/TSMC/CERN_C65LP/digital/Back_End/lef/tcbn65lp_200a/
    lef/tcbn65lp_9lmT2.lef ../SRAM/tsdn65lpa65536x16m32s_200b/
    LEF/tsdn65lpa65536x16m32s_200b_5m.lef ../SRAM/
    tsdn65lpa65536x16m32s_200b/LEF/
    tsdn65lpa65536x16m32s_200b.alef  ../SRAM/
    tsdn65lplla8192x16m16s_200a/LEF/
    tsdn65lplla8192x16m16s_200a_5m.lef ../SRAM/
    tsdn65lplla8192x16m16s_200a/LEF/
    tsdn65lplla8192x16m16s_200a.alef}
read_netlist "../output/export.phys.v"
init_design
read_def ../output/export.def
read_spef -rc_corner RC_WORST ../output/export_RC_WORST.spef
read_spef -rc_corner RC_BEST ../output/export_RC_BEST.spef
read_spef -rc_corner RC_TYP ../output/export_RC_TYP.spef
```

Listing 29: Part of script for reading power simulation

(1) DEF
DEF(Design Exchange Format) is used to store physical design information and contains design specific information.

(2) Physical gate-level netlist
The information in the physical gate-level netlist differs from that used in previ-

ous simulations in that. It also includes power ground nets and physical cells like fillers and spacers and decoupling cells.

(3) SPEF
SPEF (Standard Parasitic Exchange Format) is an IEEE standard for representing parasitic in a chip wires in ASCII format. This data includes parasitic information such as capacitances, resistances and inductances of wires. This file is generated after the P&R.

(4) Technology LEF and Macro LEF files.

(5) MMMC for process corner and mode definition.

When setting the activity scope, it should be consistent with the module name and instance name of the testbench, as shown below.

```
set ACTIVITY_SCOPE testbench_read/mipsfpganexys4_read
```

6.2 Gate Level Simulation

Gate level simulation and RTL level simulation differ in terms of the used abstraction level but also because of the timing information taken from the SDF file. Due to the introduced delays clock setup and hold violations can occur, which give rise to the generation of unknown signals, which can propagate through the complete circuits which is a huge source of errors and is difficult to solve. In the following solutions for the problems encountered during the simulation are presented.

- **Initialization of Memories**
  For prevention of problems caused by passing undefined output values of SRAMs used in the data and instruction caches to the AHB and other logic blocks, all memory arrays in both caches need to be initialized in the testbench by loading an initial file memoryinitial.txt, which contains only zero data, as shown in Listing 30.

```
initial
begin
...
testbench_read.mipsfpganexys4_read.
mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst0.MX.
load("../rtl_up/memoryinitial.txt");
...
end
```

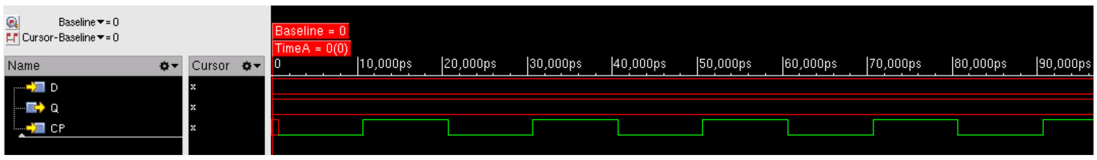
Listing 30: Resigter module mvp_register

- **Flip-Flop without Reset**
  To reduce area and power consumption flip-flops used in the MIPS Core, do not have a reset signal and can not be initialized after the power-up, as shown in Listing 31. The

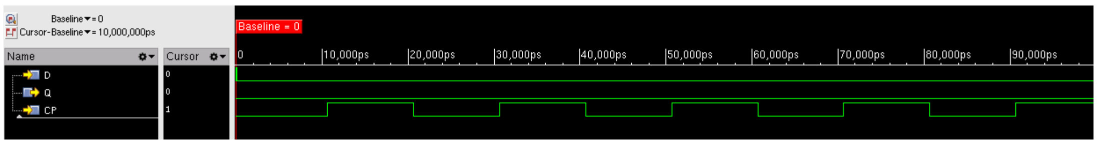
disadvantage of this configuration is that the simulation starts with undefined values stored in the flip-flops which are propagated as value x to the outputs. Simulation results of the registers before and after initialization are shown in Figure 6.1.

```
‘include "m14k_const.vh"
module mvp_register (
        q,
        clk,
        d
);
// synopsys template
parameter WIDTH = 1;
output  [WIDTH-1:0] q;
reg     [WIDTH-1:0] q;
input               clk;
input   [WIDTH-1:0] d;
always @(posedge clk)
begin
q <= #‘M14K_FDELAY d;
‘ifdef MIPS_SIMULATION
‘ifdef M14K_XCHECK
if ((clk === 1’bx) && (q !== d))
q <= #‘M14K_FDELAY {WIDTH{1’bx}};
‘endif
‘endif
end
endmodule
```

Listing 31: Resigter module mvp_register



(a)



(b)

Figure 6.1: Simulation result before and after initialization of the register

In order to solve flip-flops initialization, all signals are set to zero at the beginning by using the option "*-dfile*" with the command *xrun*. In addition, the value 0 is assigned to input signals enabled via the "*-dfile*" option through the Tcl *deposit* command. In the following the relevant options are given. The contents of two files are shown in Listing 32,33, where the specified object is defined by the option "*-scope*," while "*-depth all*" indicates all level of hierarchy. The option "*-if_x*" is used to initialize input signals which have a value of "x," the initial value zero is set by the option "*-allzero*". The command *deposit* is similar to the command *force*, but instead of forcing a permanent change of the signal, it allows changes when new input signal values are assigned.

```
xrun ... -dfile ../sim/dfile.txt -input ../sim/scripts/
   dfile.tcl...
```

Listing 32: Options for input initialization

```
MODULE *
PORTS IN *
ENDMODULE
```

Listing 33: dfile.txt

```
deposit -scope testbench_read.mipsfpganexys4_read -depth all
   -if_x -allzero -verbose debug
```

Listing 34: dfile.tcl

- **Clock Tree Timing Violation**
  Signal initialization affects especially the clock tree. As can be seen in Figure 6.2, the clock is transferred through multiple inverters to each cell controlled by the clock. As is shown in Figure 6.3 due to the short delay in the inverter, the inversion of signal CTS_58 from 1 to 0 can not be completed and sent to CTS_57 in time, which results into a timing violation in Listing 35 which gives rise to undefined signals. In order to solve this problem, the initial value of all signals in the clock tree have to be found for example by simulating without timing back-annotation. These values have to be introduced in the script dfile.tcl, in which the value of each clock signal is adjusted after 0 ns by means of the command *deposit*, as is shown in Listing 36. E.g. From the simulation result, it is clear that the clock signal CTS_57 should be set to zero at the beginning.

```
Warning!  Timing violation
$width( posedge CLKA:120 PS,  : 610 PS,  0.51 : 510 PS );
File: ../SRAM/
tsdn65lpa65536x16m32s_200b/VERILOG/
tsdn65lpa65536x16m32s_200b_ff1p32v0c.v, line = 588
Scope: testbench.mipsfpganexys4.mipsfpga_sys_mipsfpga_ahb_
```

```
mipsfpga_ahb_ram_reset_
sram_interface_15_TSDN65LPA65536X16M32S_2
Time: 610 PS
```

Listing 35: Timing violation during gate-level-simulation

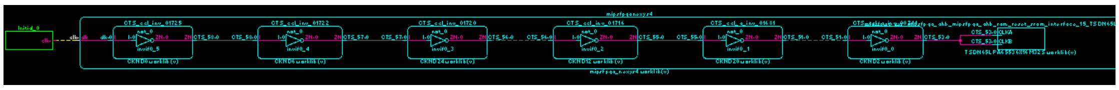


Figure 6.2: Schematic trace of CLKA in AHB SRAM

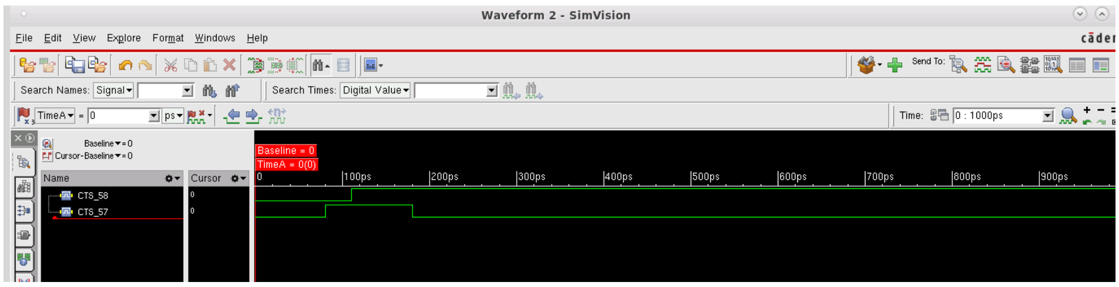


Figure 6.3: Clock with glitch

```
deposit testbench.mipsfpganexys4.CTS_57 0 -after 0 ns
    -absolute
```

Listing 36: Modification of clock signal CTS_57 through the command deposit

- **X-propagation after Initialization**
  Although the uninitialized inputs and warnings about clocks have been resolved, there are still unknown signals in the simulation that lead to the inability to properly complete the desired result in the simulation. The following methods can solve the X-propagation problem that still exists.

  (1) Address Signal Tracing
  As shown in Figure 4.10, it is clear that the address of the SRAM in the interface AHB is derived from the signal *edp_abus_e[31:0]* in the module execution data path, and the simulation result of memory *sram_interface_15* in Figure 6.4 below shows that the address signal is in an unknown state shortly after 2000 ns. In order to detect the reason, the signal source has to be traced in the netlist. Two signals are finally identified which are used to select signals in a multiplexer module.

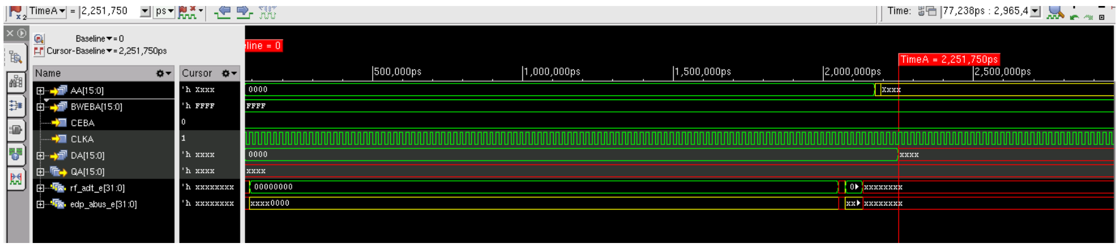


Figure 6.4: Signals from ahb_ram_reset.sram_interface_15 and address source signals

i. edp.mpc_aselres_e

From the following instantiation of the signal *edp_abus_e* and the Verilog description of the 2:1 MUX module it can be expected that *rf_adt_e[28]* is passed to *edp_abus_e[28]* when the signal *mpc_aselres_e* is zero. However, the schematic shown in Figure 6.5 does not match the Verilog description and the instantiation, since the selection signal *mpc_aselres_e* whose value is zero is passed to the signal *FE_OFN51724_n_4753* through a buffer, then the output value of the AND gate between the signal *FE_OFN51724_n_4753* and *FE_OFN52393_rf_adt_e_28* is constant zero. Thus, the signal *rf_adt_e[28]* can not be selected and the value of the signal *edp_abus_e[28]* is always zero. In order to enable the selection function, the selection signal *mpc_aselres_e* needs to be flipped from 0 to 1 by using an instantiation with another form of 2 to 1 multiplexer, which is described in more detail in the following text about the modification of the Verilog description of the signal *edp_abus_e*.

```
mvp_mux2 #(.WIDTH(16)) _edp_abus_e_31_16_        (.y(edp_abus_e[31:16]),.sel(mpc_aselres_e), .a(rf_adt_e[31:16]), .b(preabus_e[31:16]));
```

Listing 37: 2:1 MUX for the signal edp_abus_e

```
`include "m14k_const.vh"
module mvp_mux2(
        y,
        sel,
        a,
        b
);
parameter WIDTH = 1;
output  [WIDTH-1:0] y;
reg     [WIDTH-1:0] y;
input               sel;
input   [WIDTH-1:0] a;
input   [WIDTH-1:0] b;
always @(sel or a or b)
begin
        case(sel)
```

```verilog
                        1'b0  :             y  =  a;
                        1'b1  :             y  =  b;
                        default:            y  =  {WIDTH{1'bx}};
            endcase
end
endmodule
```

Listing 38: Verilog description of the 2:1 MUX module

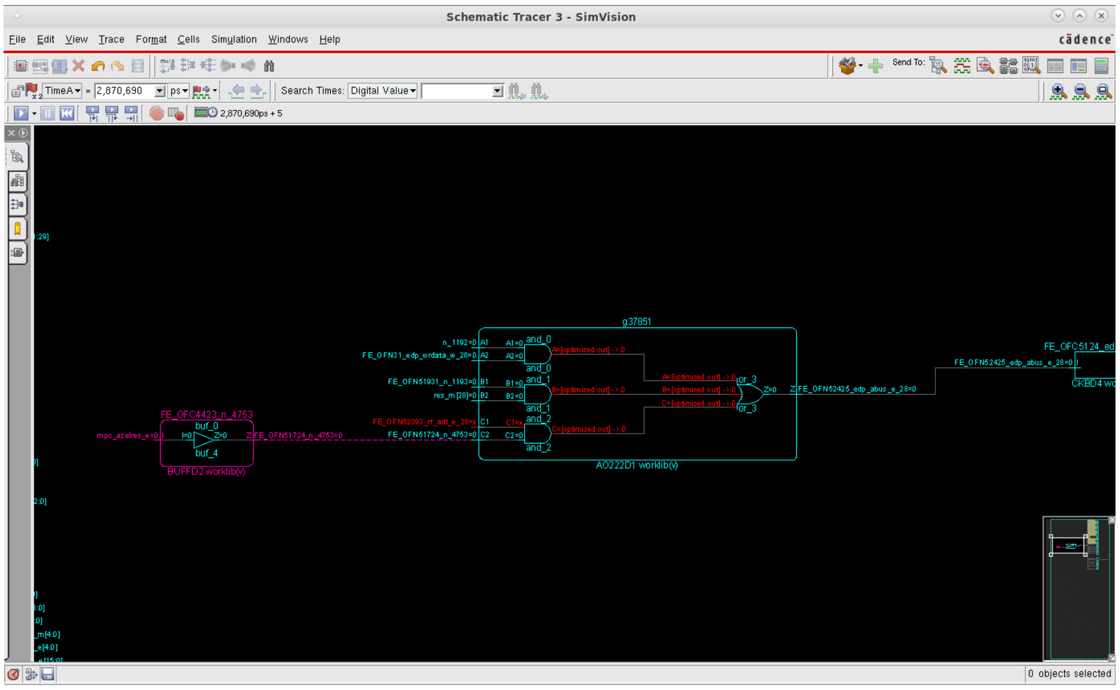


Figure 6.5: Schematic of the signal edp_abus_e[28] and rf_adt_e[28]

From the description of the decoder and the selector in the AHB module given in chapter 2, it is clear that the above errors lead to wrong address signals and ultimately wrong decisions in addressing the SRAM.

ii. edp.n_9321

The signal *edp.n_9321* is used as the selection signal for the low bit of the signal *edp_abus_e*. It is known from the schematic in Figure 6.6 that the signal is undriven, which affects the output of the AND gate and therefore causes the X-propagation.

(2) Modification of the Verilog description of the signal edp_abus_e [31:0]

From the above analysis, it can be concluded that the signal *edp_abus_e* can not get the correct value, even if these two signals are forced to 0 or 1 by the command deposit or force. The critical point is that the selection signal of the 2:1 MUX is always zero, although the value zero is in accordance with the logical description of the 2:1 MUX

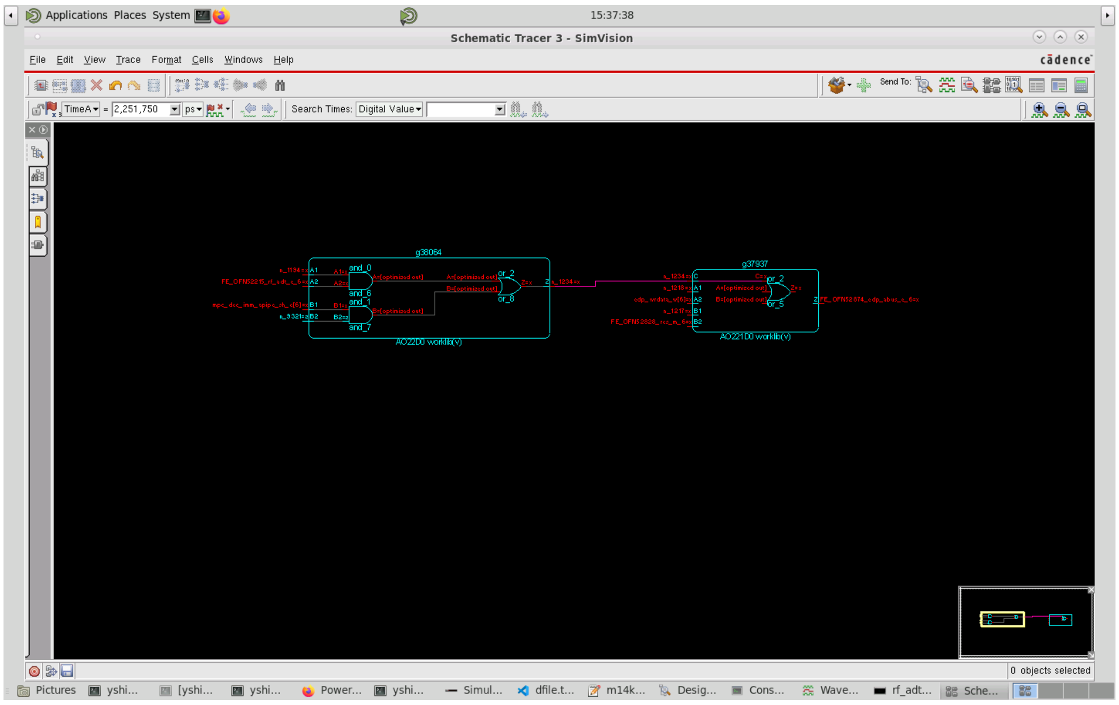


Figure 6.6: Schematic of the signal n_9321

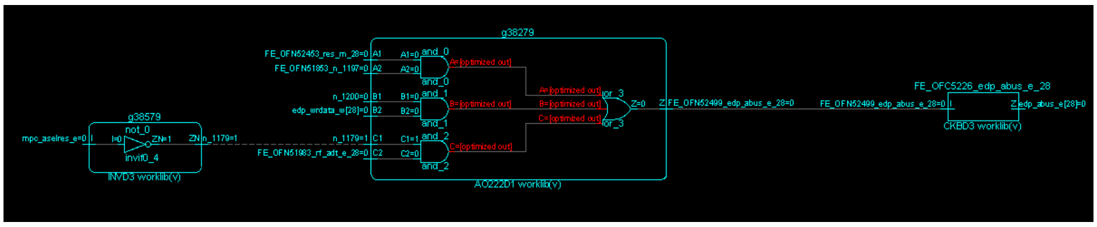
module, as described in Listing 37 for the module *mvp_mux2*, the signals corresponding to "a" is selected when all selection signals are zero during simulation. However, in the gate-level simulation an AND gate is part of the multiplexer and as a result the selection signals need to be inverted before they are assigned to the AND gate input. In order to achieve the correct operation of the signal selection, the 2:1 MUX instantiations associated with the signal *edp_abus_e* are modified using assign statements without affecting the functional simulation. The function of the respective logic is described in the following Listing in the comment statement.

```

mvp_mux2 #(.WIDTH(16)) _edp_abus_e_31_16_
(.y(edp_abus_e[31:16]),.sel(mpc_aselres_e), .a(rf_adt_e[31:16]
    ), .b(preabus_e[31:16]));
//assign edp_abus_e[31:16]= mpc_aselres_e ?
//preabus_e[31:16] : rf_adt_e[31:16];

mvp_mux2 #(.WIDTH(16)) _abus_noinsv_e_15_0_
(.y(abus_noinsv_e[15:0]),.sel(mpc_aselres_e), .a(rf_adt_e[15:0
    ]), .b(preabus_e[15:0]));
//assign abus_noinsv_e[15:0] = mpc_aselres_e ?
//preabus_e[15:0] : rf_adt_e[15:0];

mvp_mux2 #(.WIDTH(5)) _abus_noinsv_imm_e_4_0_
```

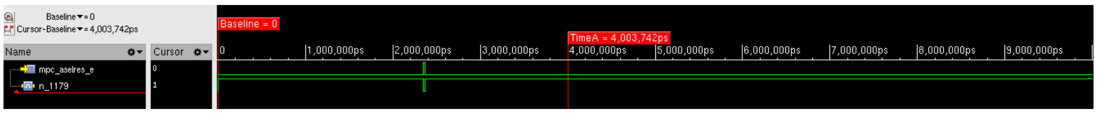
```
(.y(abus_noinsv_imm_e[4:0]),.sel(mpc_dec_imm_rsimm_e), .a(
    abus_noinsv_e[4:0]),.b( mpc_dec_imm_apipe_sh_e[4:0]));
//assign abus_noinsv_imm_e[4:0] = mpc_dec_imm_rsimm_e ?
//mpc_dec_imm_apipe_sh_e[4:0] : abus_noinsv_e[4:0];

mvp_mux2 #(.WIDTH(5)) _edp_abus_e_4_0_
(.y(edp_abus_e[4:0]),.sel(mpc_dec_insv_e), .a(
    abus_noinsv_imm_e[4:0]), .b(dsp_dspc_pos_e[4:0]));
//assign edp_abus_e[4:0] = mpc_dec_insv_e ?
//dsp_dspc_pos_e[4:0] : abus_noinsv_imm_e[4:0];

mvp_mux2 #(.WIDTH(11)) _edp_abus_e_15_5_
(.y(edp_abus_e[15:5]),.sel(mpc_dec_imm_rsimm_e), .a(
    abus_noinsv_e[15:5]),.b( mpc_dec_imm_apipe_sh_e[15:5]));
//assign edp_abus_e[15:5] = mpc_dec_imm_rsimm_e ?
//mpc_dec_imm_apipe_sh_e[15:5] : abus_noinsv_e[15:5];
```

Listing 39: Verilog description of the selection source signals for edp_abus_e

After confirmation by functional simulation, synthesis and P&R, the modified gate-level circuit is simulated again, and the following results are obtained. Comparing the result in Figure 6.5 with the new schematic and simulation result in Figure 6.7, it can be seen that the selection signal *mpc_aselres_e* is first transferred through an inverter and the value is switched from 0 to 1, so that *rf_adt_e [28]* can be selected as expected and transferred to *edp_abus_e [28]*. Another example is the selection signal *n_9321* where



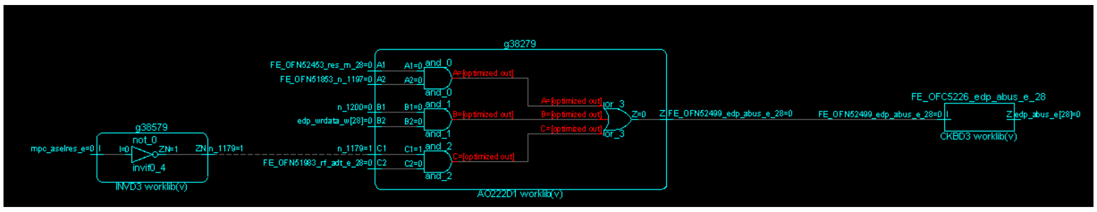
(a)



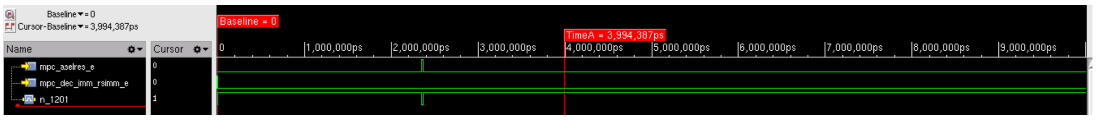
(b)

Figure 6.7: New schematic a) and simulation b) for the selection signal of edp_abus_e[28]

the original design gives rise to the value 'z'. As is shown in Figure 6.8, this signal is replaced by an other signal *mpc_dec_imm_rsimm_e* as described in the Verilog statement before, and the selection signal *n_1179* of the signal *rf_adt_e [4]* originates from an inversion of the signal *mpc_aselres_e*, which meets the expectation in post-layout simulation.



(a)



(b)

Figure 6.8: New schematic a) and simulation b) for the selection signal of edp_abus_e[4]

6.2.1 Gate-Level Simulation Results

After correction of the above errors, the GLS achieves the same result as the functional simulation. When asserting the simulation command '*xrun*' mentioned in chapter 6.1 the options for the initial flip-flop and input values and the clock tree signals discussed in section 6.2 need to be added, as shown in the following Listing.

```
xrun -sv -gui -l ../sim/logs/run_post_pnr_sim.log -access +rwc
../output/export.v ../tcbn65lp.v ${PKGS_TB_NEXYS_READ}
${PKGS_RTL_memory_tb}
-timescale 1ns/10ps -mess -ntc_level 2 -sdf_nopathedge -clean
-dfile ../sim/dfile.txt -input ../sim/scripts/dfile.tcl -clean
-sdf_cmd_file ../sim/scripts/sdf_typ.cmd -clean -input
../sim/scripts/export_saif_shm_typ.tcl
```

Listing 40: New xrun command for gate level simulation

- **Annotated Percentage**
  The annotation reaches 96.33%. So the timing information is almost completely annotated to the design and the timing checks are finished.

```
SDF statistics:
No. of Pathdelays = 219107  No. of Disabled Pathdelays = 0
          Annotated = 96.33% (211066/219107)
No. of Tchecks     = 132353  No. of Disabled Tchecks     = 0
          Annotated = 89.88% (118957/132353)
```

Listing 41: Annotated percentage

- **GLS result for the memory centric program**
  The GLS result of the initial program about the first read/write instructions *lw*, *sw* is obtained as shown below.

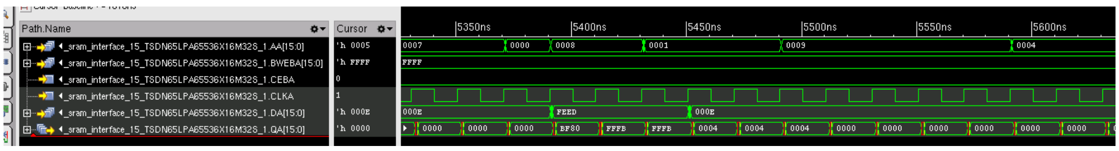

Path.Name
Cursor
_sram_interface_15_TSDN65LPA65536X16M32S_1.AA[15:0]
_sram_interface_15_TSDN65LPA65536X16M32S_1.BWEBA[15:0]
_sram_interface_15_TSDN65LPA65536X16M32S_1.CEBA
_sram_interface_15_TSDN65LPA65536X16M32S_1.CLKA
_sram_interface_15_TSDN65LPA65536X16M32S_1.DA[15:0]
_sram_interface_15_TSDN65LPA65536X16M32S_1.QA[15:0]
5350ns
5400ns
5450ns
5500ns
5550ns
5600ns

Figure 6.9: GLS Result for lw and sw instructions

- **GLS Result for the register centric design**
  The GLS result for Address cycling is shown below, and the result is the same as in the functional simulation.

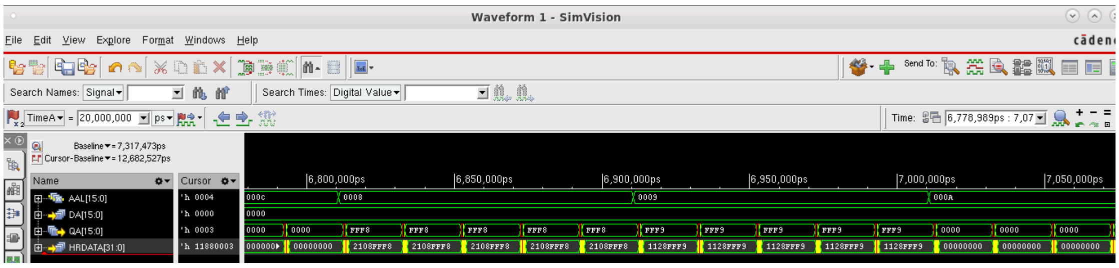


Figure 6.10: GLS Result for register-centric program

6.3 Power Simulation

This section describes the average power consumption, which can be divided into static power and dynamic power. The static power refers to the power dissipation caused by leakage currents in the transistor, and can also be called leakage power. Dynamic power refers to the power consumption caused by switching activity and can be divided into internal power caused by charging and discharging of internal capacitors and switching caused by external load capacitor charging and discharging [13]. Power consumption can be calculated manually or evaluated automatically by tools like Voltus.

- **Manual Calculation**
  According to the different types of power consumed in standard cells and macro components, the power consumption is estimated by the combination of the three contributions, as shown below.

$$P_{total} = P_{leakage} + P_{switching} + P_{internal} \tag{6.1}$$

  The three types of power consumption are described as follows.

i. Leakage Power
Since the leakage power is state-dependent, there are different values for different combinations of inputs, and the probability of each state is different. The calculation of the weighted sum of multiple states is based on the leakage power value and the probability of input patterns. For example, in the liberty description of the standard cell "AND2D0" various input conditions are described.

```
cell (AN2D0) {
area : 2.16;
cell_footprint : "an2d1";
...
leakage_power () {
value : 0.004594;
related_pg_pin : VDD;}
leakage_power () {
value : 0.003626;
when : "!A1!A2!Z";
related_pg_pin : VDD;}
leakage_power () {
value : 0.004183;
when : "!A1A2!Z";
related_pg_pin : VDD;}
leakage_power () {
value : 0.005130;
when : "A1!A2!Z";
related_pg_pin : VDD;}
leakage_power () {
value : 0.005439;
when : "A1A2Z";
related_pg_pin : VDD;}
```

In following, the complete clause set is calculated. The probability of all clauses should sum up to 1. If it is less than 1, the clause set is incomplete. If it is greater than 1, there is an error in the library data. The equation for calculating the complete clause set is given as.

$$P_{leakage} = \sum_{i=1}^{4} (p_i * prob_i) \tag{6.2}$$

where p is the leakage value for each condition and prob is the probability for each input value.
The possible leakage power for the cell in the above example is shown in Equation

6.3.

$$
\begin{aligned}
&P_{leakage}(AN2D0) = \sum_i (p_i * prob_i) \\
&= 0.004594 * prob(1) + 0.003626 * prob(2) \\
&+ 0.004183 * prob(3) + 0.005130 * prob(4) \\
&Prob_{total} = \sum_{i=1}^{4} probi = 1
\end{aligned}
\tag{6.3}
$$

Definitions for use with incomplete clause sets are shown below, the first of which can replace the missing condition with the generic leakage value, while the other is without the generic leakage value.

$$
\begin{aligned}
P_{incom\_gene} &= P_{leakage} + p_{generic} * (1 - prob_{other}) \\
P_{incom\_without\_gene} &= \frac{P_{leakage}}{Prob_{total}}
\end{aligned}
\tag{6.4}
$$

ii. Switching Power
The switching power is related to the supply voltage, load capacitance and transition density, and is calculated as given below.

$$
P_{switching} = \frac{1}{2} CV^2 D \tag{6.5}
$$

where:
C = load capacitance
V = supply voltage
D = transition density

Transition density results from product of A(nodal activity or switching activity) and f(Frequency).

iii. Internal Power
The internal power can also be divided into pin power and arc power, both of which are state-dependent. The former relies on the energies represented in the "when" clause function and signal activity, and since the probability of a logic value of 0 or 1 varies from input to input, it needs to be taken into account in the calculation of energy. The latter is described in an energy table of the cell based on timing arcs. Then the total internal power is the sum of both.

∗ input pins
For a port of multiple inputs, the energy value is calculated as follows.

$$
E_{in} = \frac{1}{2} \frac{\sum_i Prob_i * (E_{rise} + E_{fall})}{\sum_i Prob_i} \tag{6.6}
$$

where:

E$_{in}$ = input energy
Prob$_i$ = probability that the value of the input i is 1 or 0
E$_{rise}$ = energy in rise transition
E$_{fall}$ = energy in fall transition

The power calculation needs to consider the transition density, which has been described before. The following is the equation of the input pins power.

$$\begin{aligned} P_{internal\_in} &= \sum_i E_{in} * AF_i \\ &= \sum_i E_{in} * D_i \end{aligned} \tag{6.7}$$

The input port energy value A1 of the AND gate cell AN2D0 can be found in the .lib file below. The attribute index_1 indicates the rise transition time and fall transition time and the attribute values shows the energy value of corresponding time. The unit defined in this .lib file is ns and pJ, respectively.

```
  pin(A1) {
direction : input;
related_ground_pin : VSS;
related_power_pin : VDD;
capacitance : 0.0006718;
rise_capacitance : 0.0006718;
fall_capacitance : 0.0006156;
internal_power () {
when : "!A2&!Z";
related_pg_pin : VDD;
rise_power (passive_power_template_7x1_0) {
index_1 ("0.0066,0.0174,0.0391,0.0825,
0.1692,0.3427,0.6897");
values ( \
"-0.0003695,-0.0003834,-0.0003892,-0.0003912,
-0.0003931,-0.0003958,-0.0004072" \
);
}
fall_power (passive_power_template_7x1_0) {
index_1 ("0.0066,0.0174,0.0391,0.0825,
0.1692,0.3427,0.6897");
values ( \
"0.0004737,0.0004847,0.0004883,0.0004912,
0.0004919,0.0004917,0.0004915" \
);
}
}
```

* output pins
  The calculation of the power at the output considers two energy values for rising- and falling transitions, the formula for calculating the output power is as follows.

$$P_{internal\_out} = \frac{1}{2}(E_{rise} + E_{fall})D_{out} \tag{6.8}$$

  The table example from the .lib file for the output energy of the cell AN2D0 can be viewed below. The switching power can also be calculated accordingly, where the floating values of the attribute index_1 refer to the input transition time and the floating values of the attribute index_2 refer to the total output net capacitance.

```
pin(Z) {
direction : output;
power_down_function : "!VDD+VSS";
function : "(A1␣A2)";
related_ground_pin : VSS;
related_power_pin : VDD;
max_capacitance : 0.02955;
internal_power () {
related_pin : "A1";
related_pg_pin : VDD;
rise_power (power_template_7x7_0) {
index_1 ("0.0066,0.0174,0.0391,0.0825,
0.1692,0.3427,0.6897");
index_2 ("0.00053,0.00099,0.00191,0.00375,
0.00744,0.01481,0.02955");
values ( \
"0.001467,0.00148,0.001506,0.001529,
0.001544,0.001554,0.001569",\
);
}
```

- **Automatic Calculation**
  The simulation based power consumption through the EDA tools such as Cadence Voltus is chosen as the main method in the present case to reduce the calculation cost for the complex netlist. The switching activity interchange format mentioned before is introduced to the power simulation to get accurate switching information. In addition to the reduced cost also calculation accuracy and generality are additional advantages [14].

6.3.1 Power Simulation Results and Analysis

This subsection focuses on comparing power simulation based on two different programs.

- **Power Consumption of lw and sw Instructions**
  First the power simulation result of the simulation using a memory centric program with many read and write instructions is presented graphically by a color code in the layout in Figure 6.11. It can be seen that the memory blocks are marked in red which according to the legend in the left corresponds to the highest power value. The detailed

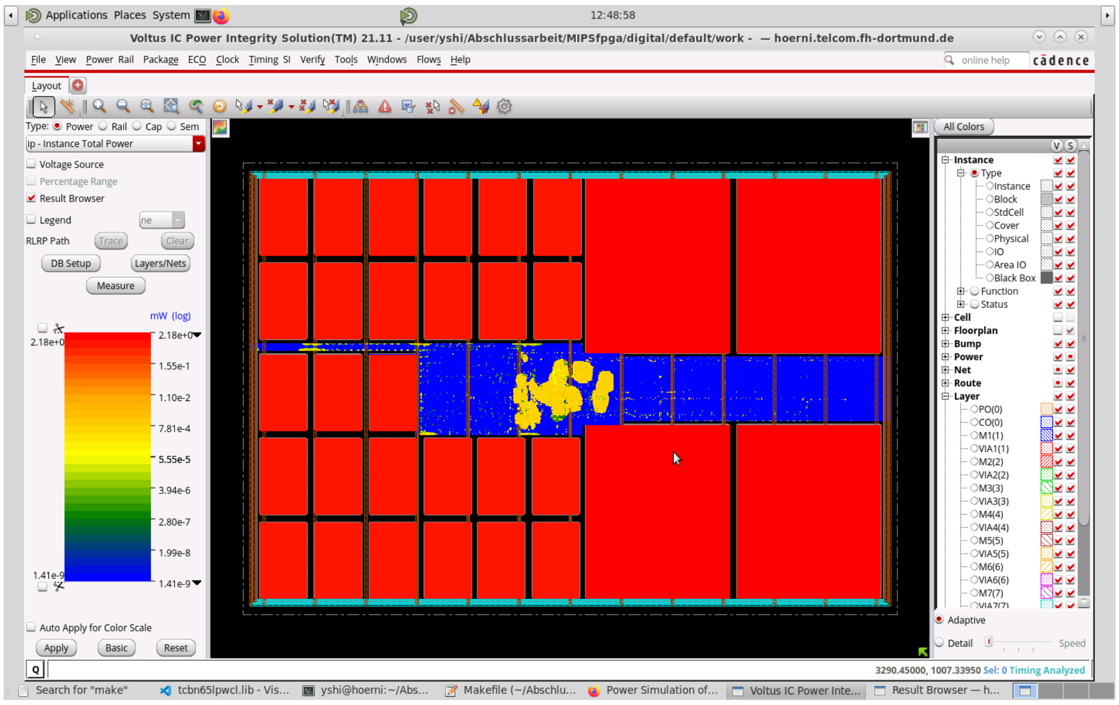
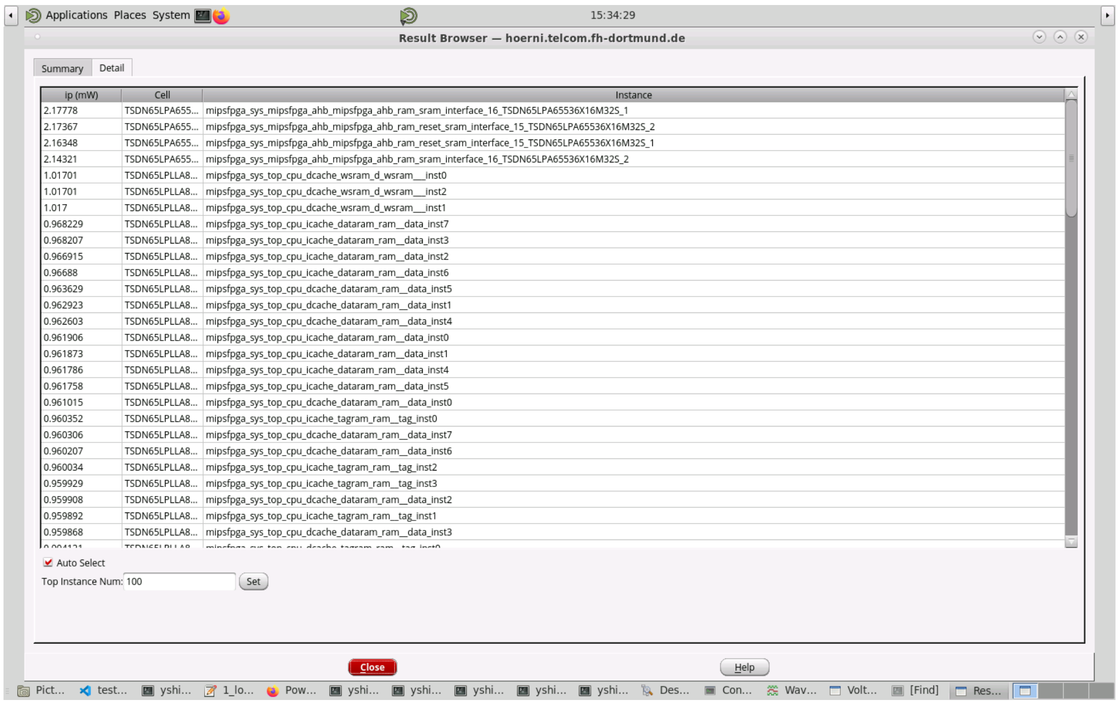


Figure 6.11: Power consumption layout for lw&sw instructions

total power consumption of each cell can be extracted by analysis of the database file *statPower.db*, from which the results of the graphical representation in the layout view are confirmed. The *sram_interface_16* consumes the highest value of 2.17778mW is . Since the executed instructions are mainly *lw* and *sw*, the two other SRAM modules of the cell *sram_interface_15* also consume 2.17367nw and 2.16348nW, respectively, as shown in Figure 6.12.


Result Browser — hoerni.telcom.fh-dortmund.de

ip (mW)	Cell	Instance
2.17778	TSDN65LPA655...	mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_sram_interface_16_TSDN65LPA65536X16M32S_1
2.17367	TSDN65LPA655...	mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_sram_interface_15_TSDN65LPA65536X16M32S_2
2.16348	TSDN65LPA655...	mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_sram_interface_15_TSDN65LPA65536X16M32S_1
2.14321	TSDN65LPA655...	mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_sram_interface_16_TSDN65LPA65536X16M32S_2
1.01701	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_wsram_d_wsram___inst0
1.01701	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_wsram_d_wsram___inst2
1.017	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_wsram_d_wsram___inst1
0.968229	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst7
0.968207	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst3
0.966915	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst2
0.96688	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst6
0.963629	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst5
0.962923	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst1
0.962603	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst4
0.961906	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst0
0.961873	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst1
0.961786	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst4
0.961758	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst5
0.961015	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst0
0.960352	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_tagram_ram__tag_inst0
0.960306	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst7
0.960207	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst6
0.960034	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_tagram_ram__tag_inst2
0.959929	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_tagram_ram__tag_inst3
0.959908	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst2
0.959892	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_tagram_ram__tag_inst1
0.959868	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst3

Auto Select ✓

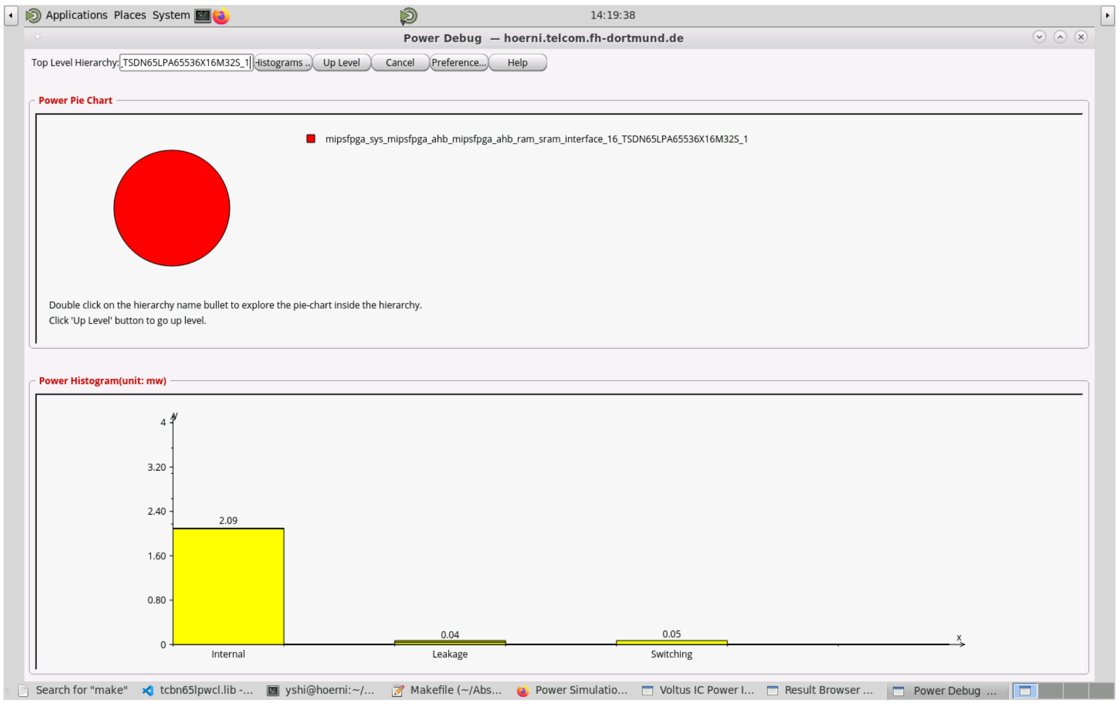
Top Instance Num: 100

Figure 6.12: Cell with highest power dissipation (lw&sw)

The histogram in Figure 6.13 shows that the power consumption of the cell with the highest contribution is dominated by internal power, which has the value of 2.09mW, followed by switching power with a value of 0.05mW while leakage power has the lowest contribution of 0.04mW. From the overall distribution of the three types of power it can be clearly seen through the following histogram, that the main power contribution is also internal power, which is 35.40mW, followed by the switching power, which consumes 1.49mW, while leakage power is again the lowest contribution with a value of 0.3mW. By means of the power simulation also the influence of clock gating can be observed. In the listing shown below the internal power consumption of the sequential cells used in the design e.g. flip-flops are shown without (old) and with (new) clock-gating enabled. As can be seen from the report internal power consumption is reduced by 3mW by activating clock-gating during implementation.

```
Group                Internal    Switching      Leakage        Total
                     Power       Power          Power          Power
-----------------------------------------------------------------------

Sequential(old)     3.838      0.01408        0.002477       3.854
Sequential(new)     0.8914     0.01439        0.002473       0.9083
```

- **Power Consumption of the Register-Centric Program**



Double click on the hierarchy name bullet to explore the pie-chart inside the hierarchy.
Click 'Up Level' button to go up level.

Figure 6.13: Three type of power consumption
cell sram_interface_16_TSDN65LPA65536X16M32S_1 (lw&sw)

Next the result of the power consumption for the simulation using a register centric program without any data memory accesses is presented. It is obvious that the power consumption of the SRAMs connected to the AHB interface is lower than the result for the program having many lw&sw instructions. However the power consumption in the caches, especially in the data cache, is higher than before, while the instruction cache power consumption is lower. The total internal power consumption as shown in Figure 6.16 is 36.12mW, which is higher than the consumption of the load/read word instructions.

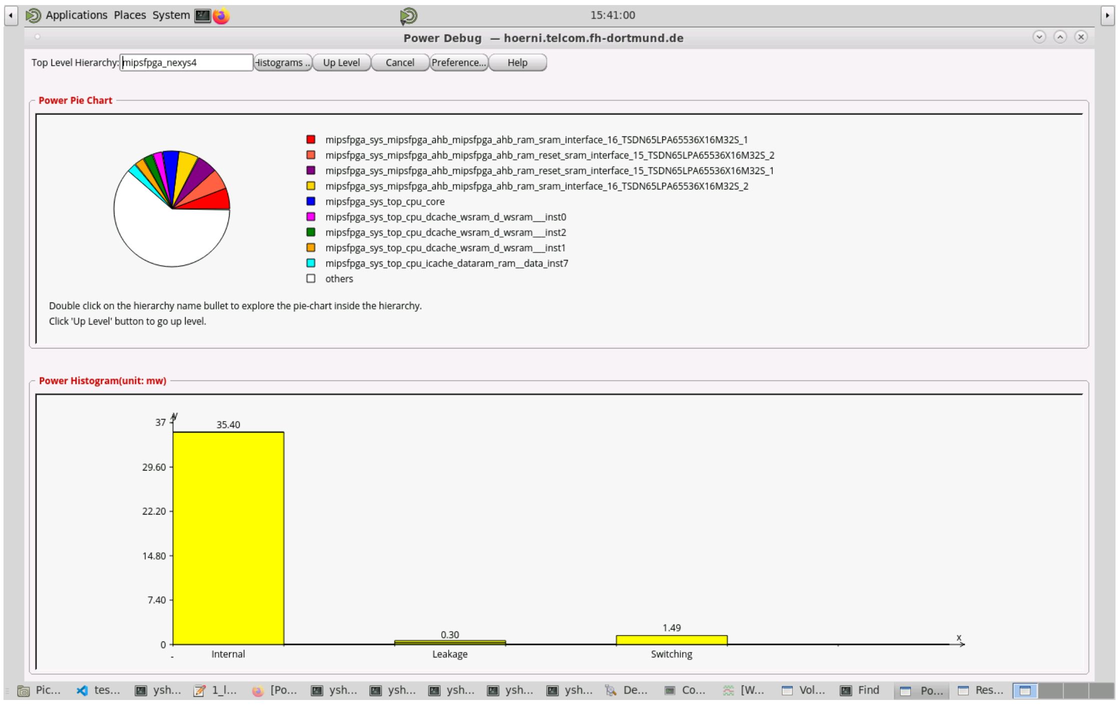
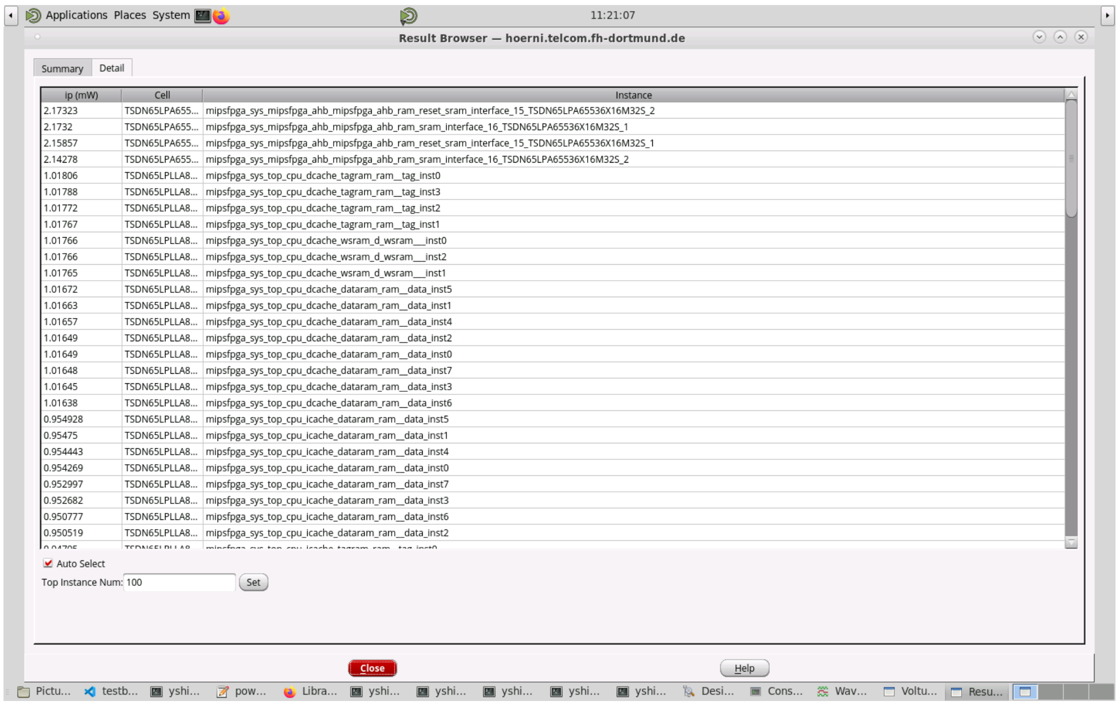

Applications Places System
15:41:00
Power Debug — hoerni.telcom.fh-dortmund.de
Top Level Hierarchy: mipsfpga_nexys4
Histograms ..
Up Level
Cancel
Preference...
Help
Power Pie Chart
🟥 mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_sram_interface_16_TSDN65LPA65536X16M32S_1
🟥 mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_sram_interface_15_TSDN65LPA65536X16M32S_2
🟪 mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_sram_interface_15_TSDN65LPA65536X16M32S_1
🟨 mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_sram_interface_16_TSDN65LPA65536X16M32S_2
🟦 mipsfpga_sys_top_cpu_core
🟪 mipsfpga_sys_top_cpu_dcache_wsram_d_wsram__inst0
🟩 mipsfpga_sys_top_cpu_dcache_wsram_d_wsram__inst2
🟧 mipsfpga_sys_top_cpu_dcache_wsram_d_wsram__inst1
🟦 mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst7
⬜ others
Double click on the hierarchy name bullet to explore the pie-chart inside the hierarchy.
Click 'Up Level' button to go up level.
Power Histogram(unit: mw)
37
29.60
22.20
14.80
7.40
0
35.40
0.30
1.49
Internal
Leakage
Switching
x

Figure 6.14: Three types of power consumption for every hierarchy (lw&sw)


Applications Places System
11:21:07
Result Browser — hoerni.telcom.fh-dortmund.de
Summary
Detail
Close
Help

ip (mW)	Cell	Instance
2.17323	TSDN65LPA655...	mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_sram_interface_15_TSDN65LPA65536X16M32S_2
2.1732	TSDN65LPA655...	mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_sram_interface_16_TSDN65LPA65536X16M32S_1
2.15857	TSDN65LPA655...	mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_sram_interface_15_TSDN65LPA65536X16M32S_1
2.14278	TSDN65LPA655...	mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_sram_interface_16_TSDN65LPA65536X16M32S_2
1.01806	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_tagram_ram__tag_inst0
1.01788	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_tagram_ram__tag_inst3
1.01772	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_tagram_ram__tag_inst2
1.01767	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_tagram_ram__tag_inst1
1.01766	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_wsram_d_wsram__inst0
1.01766	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_wsram_d_wsram__inst2
1.01765	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_wsram_d_wsram__inst1
1.01672	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst5
1.01663	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst1
1.01657	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst4
1.01649	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst2
1.01649	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst0
1.01648	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst7
1.01645	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst3
1.01638	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_dcache_dataram_ram__data_inst6
0.954928	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst5
0.95475	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst1
0.954443	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst4
0.954269	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst0
0.952997	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst7
0.952682	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst3
0.950777	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst6
0.950519	TSDN65LPLLA8...	mipsfpga_sys_top_cpu_icache_dataram_ram__data_inst2

[✓] Auto Select

Top Instance Num: 100 Set

Figure 6.15: Cell with highest power dissipation (register-centric)

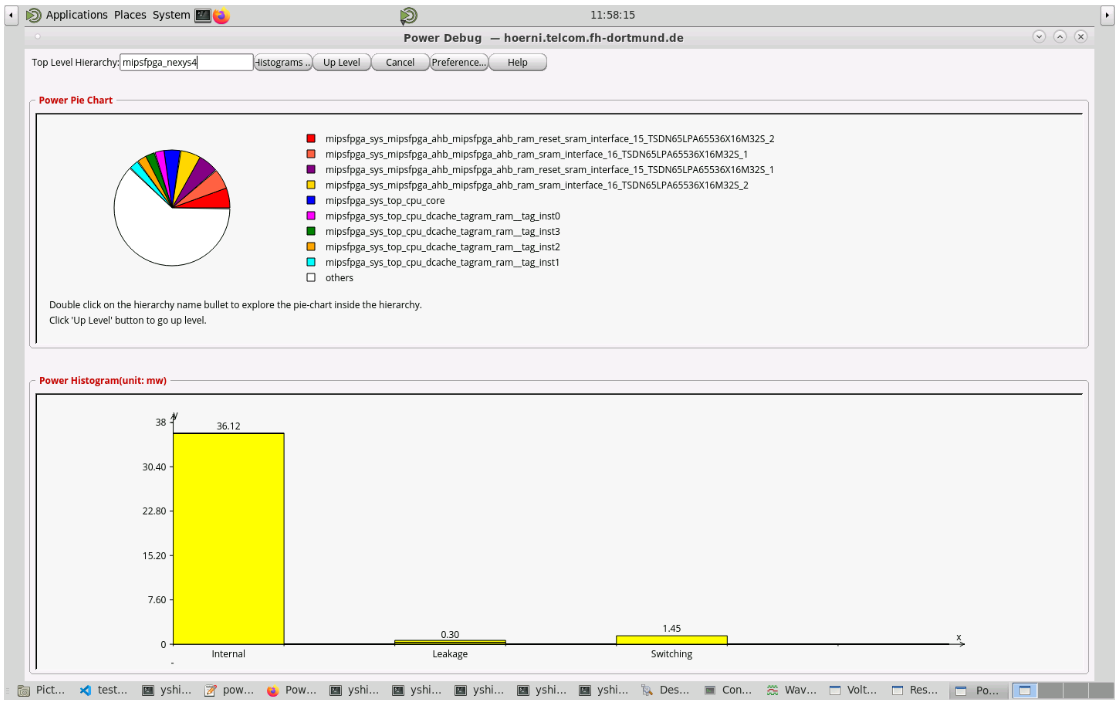

Applications Places System
11:58:15
Power Debug — hoerni.telcom.fh-dortmund.de
Top Level Hierarchy: mipsfpga_nexys4
Histograms ..
Up Level
Cancel
Preference...
Help
Power Pie Chart
🟥 mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_sram_interface_15_TSDN65LPA65536X16M32S_2
🟥 mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_sram_interface_16_TSDN65LPA65536X16M32S_1
🟪 mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_reset_sram_interface_15_TSDN65LPA65536X16M32S_1
🟨 mipsfpga_sys_mipsfpga_ahb_mipsfpga_ahb_ram_sram_interface_16_TSDN65LPA65536X16M32S_2
🟦 mipsfpga_sys_top_cpu_core
🟪 mipsfpga_sys_top_cpu_dcache_tagram_ram__tag_inst0
🟩 mipsfpga_sys_top_cpu_dcache_tagram_ram__tag_inst3
🟧 mipsfpga_sys_top_cpu_dcache_tagram_ram__tag_inst2
🟦 mipsfpga_sys_top_cpu_dcache_tagram_ram__tag_inst1
⬜ others
Double click on the hierarchy name bullet to explore the pie-chart inside the hierarchy.
Click 'Up Level' button to go up level.
Power Histogram(unit: mw)
y
38
30.40
22.80
15.20
7.60
0
x
36.12
0.30
1.45
Internal
Leakage
Switching
Pict... test... yshi... pow... Pow... yshi... yshi... yshi... yshi... yshi... Des... Con... Wav... Volt... Res... Po...

Figure 6.16: Three types of power for every hierarchy (register-centric)

From the above two power consumption results, it is clear that the power consumption of the SRAMs connected to the AHB interface is higher when *lw* and *sw* instructions are used than when no lw and sw instructions are used, which is in line with the expected result. But surprisingly the overall macro power consumption of the memory-centric program is lower than of the register-centric program, mainly due to the fact that the power consumption in the data cache is relatively low and the macro internal power is less by 0.74mW, as shown in the table below.

```
Group              Internal    Switching    Leakage     Total
                   Power       Power        Power       Power
----------------------------------------------------------------
macro(lw&sw)       34.15       0.1371       0.2897      34.57
macro(cycling)     34.89       0.1371       0.2897      35.32
```

7 Conclusions

This thesis provides a basic understanding of the MIPS instruction set and the internal operation of the MIPS core. It also introduces the results of functional simulation for different instructions. A problem with undefined signals marked with X is encountered during the GLS, as described above. After using a new form of 2 to 1 multiplexer for the signal edp_abus_e and setting the input initial value to zero, the problem could be solved. This result is however remarkable because other parts of the logic also use the same module MUX2 and do not show any problems. The power consumption of the SRAMs and other logic cells, and registers in the core is also slightly different than the expected results. The deviation is mainly caused by the data cache and a deeper understanding of how the cache works would be necessary to get a valid explanation for the observed behavior.

REFERENCES

[1] MIPS Tech, "MIPS32® microAptiv™ UP Processor Core Family Software User's Manual(MD00942) 2014," [Online]. Available: https://www.mips.com/downloads/mips32-microaptiv-up-processor-core-family-software-users-manual/

[2] Harris, S., Owen, R., Sedano, E., et al.: 'MIPSfpga: hands-on learning on a commercial soft-core'. 11th European Workshop on Microelectronics Education (EWME), Southampton, England, May 2016, pp. 1–5

[3] Dogan Ibrahim, PIC32 Microcontrollers and the Digilent Chipkit: Introductory to Advanced Projects, 1st ed. Newton, MA: Newnes, 2015

[4] Michael Karagounis, "Studie zur Nutzung eines Mikroprozessor Cores am Beispiel der MIPS Prozessorarchitektur," March. 2018.

[5] Xilinx, Inc., "Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide (UG953) 2019," [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug953-vivado-7series-libraries.pdf

[6] The Altera Corporation, "Quartus II Handbook Version 9.1 Volume 1: Design and Synthesis 2009," Altera, San Jose, CA, USA, [Online]. Availavle: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/archives/quartusii_handbook_9.1.1.pdf

[7] TSMC 65nm Low Leakage Low Power Dual-Port SRAM Compiler Databook

[ref8] enus User Guide Product Version 21.1, December 2021

[8] LEF/DEF 5.8 Language Reference Product Version 5.8

[9] Synopsis Design Constraints

[10] Sridhar Gangadharan, Sanjay Churiwala. Constraining Designs for Synthesis and Timing Analysis: A Practical Guide to Synopsys Design Constraints (SDC), Springer.

[11] Introduction to Simulation History Manager (SHM) Rapid Adoption Kit (RAK)

[12] Voltus Stylus Common UI User Guide Product Version 21.11, August 2021

[13] Yehya Nasser, Jordane Lorandel, Jean-Christophe Prévotet and Maryline Hélard, "RTL to Transistor Level Power Modeling and Estimation Techniques for FPGA and ASIC: A Survey," IEEE Trans. Comput.-Aided Design Integr. Ciucuits and Syst, vol. 15, no. 3, pp. 479-492, March. 2021.

A Code for the memory TSN65LPLLDPSRAM

The following are the codes for each of the two types of memory.

```
`resetall
`celldefine
`timescale 1ns/10ps
`delay_mode_path
`suppress_faults
`enable_portfaults
`ifdef UNIT_DELAY
`define SRAM_DELAY 0.010
`endif
module TSDN65LPA65536X16M32S
  (AA,
  DA,
  BWEBA,
  WEBA,CEBA,CLKA,
  AB,
  DB,
  BWEBB,
  WEBB,CEBB,CLKB,
  QA,
  QB);
// Parameter declarations
parameter  N = 16;
parameter  W = 65536;
parameter  M = 16;
...
...
...
// Task for loading contents of a memory
task load;   //{ USAGE: initial inst.load ("file_name");
input [256*8:1] file;  // Max 256 character File Name
begin
$display ("\n%m:␣Reading␣file,␣%0s,␣into␣memory", file);
$readmemh (file, mem, 0, Nword-1);
end
endtask //}
...
...
endmodule
```

Listing 42: Part of code for TSDN65LPA65536X16M32S

```
`resetall
`celldefine
`timescale 1ns/1ps
`delay_mode_path
`suppress_faults
```

```
`enable_portfaults
`ifdef UNIT_DELAY
`define SRAM_DELAY 0.010
`endif
module TSDN65LPLLA8192X16M16S
  (AA,
  DA,
  BWEBA,
  WEBA,CEBA,CLKA,
  AB,
  DB,
  BWEBB,
  WEBB,CEBB,CLKB,
  QA,
  QB);
// Parameter declarations
parameter  N = 16;
parameter  W = 8192;
parameter  M = 13;

...
...
...
// Task for loading contents of a memory
task load;   //{ USAGE: initial inst.load ("file_name");
input [256*8:1] file;  // Max 256 character File Name
begin
$display ("\n%m:␣Reading␣file,␣%0s,␣into␣memory",
file);
$readmemb (file, mem, 0, Nword-1);
end
endtask //}
endtask //}
...
...
endmodule
```

Listing 43: Part of code for TSDN65LPLLA8192X16M16S

B Code for SRAM Interface and Memory replacement

```
module sram_interface

(
    //Signals to SRAM1 Port A (lower 16 bits)
    input       [15:0]                    AA1,
    input       [15:0]                    DA1,
    input       [15:0]                      BWEBA1,
    input                                   WEBA1,
    input                                   CEBA1,
    input                                   CLKA1,
    output      [15:0]                      QA1,
    //Signals to SRAM2 Port A (upper 16 bits)
    input       [15:0]                      AA2,
    input       [15:0]                      DA2,
    input       [15:0]                      BWEBA2,
    input                                   WEBA2,
    input                                   CEBA2,
    input                                   CLKA2,
    output      [15:0]                      QA2
);
//Combine 2x16bit from SRAM into single 32bit for the mem bridge
TSDN65LPA65536X16M32S TSDN65LPA65536X16M32S_1(
.AA(AA1),
.DA(DA1),
.BWEBA(BWEBA1),
.WEBA(WEBA1),
.CEBA(CEBA1),
.CLKA(CLKA1),
.QA(QA1),
.WEBB   (1'b0),
.CEBB   (1'b1),
.BWEBB  (16'h0000),
.CLKB   (clk),
.AB     (16'h0000),
.DB     (16'h0000),
.QB     ()

);
TSDN65LPA65536X16M32S TSDN65LPA65536X16M32S_2(
.AA(AA2),
.DA(DA2),
.BWEBA(BWEBA2),
.WEBA(WEBA2),
.CEBA(CEBA2),
.CLKA(CLKA2),
.QA(QA2),
.WEBB   (1'b0),
.CEBB   (1'b1),
.BWEBB  (16'h0000),
.CLKB   (clk),
.AB     (16'h0000),
.DB     (16'h0000),
.QB     ()
);
endmodule
```

Listing 44: SRAM interface module

```
`timescale 100ps/1ps

`include "mipsfpga_ahb_const.vh"

module mipsfpga_ahb_ram_reset
(
    input                    HCLK,
    input                    HRESETn,
    input           [ 31: 0] HADDR,
    input           [ 31: 0] HWDATA,
    input                    HWRITE,
    input                    HSEL,
    output          [ 31: 0] HRDATA
);

  wire [31:0] HADDR_d;
//H_RAM_RESET_ADDR_WIDTH=15
  wire [15:0] HRDATA_1;
  wire [15:0] HRDATA_2;

sram_interface  sram_interface_15
(
.DA1            (HWDATA[15:0]),
.AA1            ({1'b0,HADDR[(`H_RAM_RESET_ADDR_WIDTH+1):2]}),
.WEBA1          (~(HWRITE&HSEL)),
.BWEBA1         (16'h0000),
.CEBA1          (1'b0),
.CLKA1          (HCLK),
.QA1            (HRDATA_1)

.DA2            (HWDATA[31:16]),
.AA2            ({1'b0,HADDR[(`H_RAM_RESET_ADDR_WIDTH+1):2]}),
.WEBA2          (~(HWRITE&HSEL)),
.BWEBA2         (16'h0000),
.CEBA2          (1'b0),
.CLKA2          (HCLK),
.QA2            (HRDATA_2)
  );

 assign  HRDATA={HRDATA_2,HRDATA_1};
endmodule
```

Listing 45: Instantiation between memory and upper level module

```
 `include "m14k_const.vh"
module tagram_2k2way_xilinx(
        clk,
        line_idx,
        rd_str,
        wr_str,
        early_ce,
        greset,
        wr_mask,
        wr_data,
        rd_data,
        hci,
        bist_to,
        bist_from);
        /* Inputs */
        input                   clk;      // Clock
        input  [6:0]            line_idx;// Read Array Index
        input                   rd_str;  // Read Strobe
        input                   wr_str;  // Write Strobe
        input  [1:0]            wr_mask; // Write Mask
        input  [23:0]           wr_data; // Data for Tag Write
        input  [0:0]            bist_to;
        input                   early_ce;
        input                   greset;
        /* Outputs */
        output[47:0]            rd_data; // output from read
        output[0:0]             bist_from;
        output                  hci;
        assign hci = 1'b0;
        assign bist_from[0] = 1'b0;
        wire [31:0] wide_wr_data = {8'b0, wr_data};
        wire [63:0] wide_rd_data;
        wire [47:0] rd_data = {wide_rd_data[55:32], wide_rd_data[
           23:0]};

        wire    [1:0]   en;
`ifdef M14K_EARLY_RAM_CE
        assign  en = {2{early_ce}};
`else
        assign  en = {2{wr_str}} & wr_mask | {2{rd_str}};
`endif

TSDN65LPLLA8192X16M16S ram__tag_inst0 (
.WEBA   (~(wr_str && wr_mask[0])),
.BWEBA  (16'h1111),
.CEBA   (en[0]),
.CLKA   (clk),
.AA     ({6'b0,line_idx}),
.DA     (wide_wr_data[15:0]),
.QA     (wide_rd_data[15:0]),
.WEBB   (1'b0),
.CEBB   (1'b1),
.BWEBB  (16'h0000),
.CLKB   (clk),
.AB     (16'h0000),
.DB     (16'h0000),
.QB     ()
);
TSDN65LPLLA8192X16M16S ram__tag_inst1 (
.WEBA   (~(wr_str && wr_mask[0])),
.BWEBA  (16'hFFFF),
.CEBA   (en[0]),
.CLKA   (clk),
.AA     ({6'b0,line_idx}),
.DA     (wide_wr_data[31:16]),
.QA     (wide_rd_data[31:16]),
.WEBB   (1'b0),
.CEBB   (1'b1),
.BWEBB  (16'h0000),
.CLKB   (clk),
.AB     (16'h0000),
.DB     (16'h0000),
.QB     ()
);

TSDN65LPLLA8192X16M16S ram__tag_inst2 (
.WEBA   (~(wr_str && wr_mask[1])),
.BWEBA  (16'hFFFF),
.CEBA   (en[1]),
.CLKA   (clk),
.AA     ({6'b0,line_idx}),
.DA     (wide_wr_data[15:0]),
.QA     (wide_rd_data[47:32]),
.WEBB   (1'b0),
.CEBB   (1'b1),
.BWEBB  (16'h0000),
.CLKB   (clk),
.AB     (16'h0000),
.DB     (16'h0000),
.QB     ()
);
TSDN65LPLLA8192X16M16S ram__tag_inst3 (
.WEBA   (~(wr_str && wr_mask[1])),
.BWEBA  (16'hFFFF),
.CEBA   (en[1]),
.CLKA   (clk),
.AA     ({6'b0,line_idx}),
.DA     (wide_wr_data[31:16]),
.QA     (wide_rd_data[63:48]),
.WEBB   (1'b0),
.CEBB    (1'b1),
.BWEBB   (16'h0000),
.CLKB    (clk),
.AB      (16'h0000),
.DB      (16'h0000),
.QB      ()
);

endmodule
```

Listing 46: Instantiation between memory in cache and upper level module

C Testbench

```
`timescale 100ps/1ps
module testbench_read;
     reg          clk;
     reg          btnCpuReset;
     reg          btnU, btnD, btnL, btnR, btnC;
     reg   [15:0] sw;
     wire  [15:0] led;
     wire  [ 7:0] IO_7SEGEN_N;
     wire  [ 6:0] IO_7SEG_N;
     wire         JB_2;
     reg          JB_0;
     reg          JB_1;
     reg          JB_3;
     reg          JB_4;
     reg          JB_5;
     wire  [ 5:0] JB_tb;

assign     JB_tb[5:0]={JB_5,JB_4,JB_3,JB_2,JB_1,JB_0};

mipsfpga_nexys4 mipsfpganexys4_read( clk,
                            btnCpuReset,
                            btnU, btnD, btnL, btnR, btnC,
                            sw,
                            led,
                            IO_7SEGEN_N,
                            IO_7SEG_N,
                            JB_tb
                            );

initial
begin
        clk = 0;
        JB_4 = 0; JB_1 = 0; JB_0 = 0; JB_3 = 0;
        JB_5 = 1;

        forever
            #50 clk = ~ clk;

end

initial
begin
#455

testbench_read.mipsfpganexys4_read.mipsfpga_sys.mipsfpga_ahb.mipsfpga_ahb_ram_reset.sram_interface_15.TSDN65LPA65536X16M32S_1.MX.load("../rtl_up/initfiles/3_Switches&LEDs/ram_reset_init_low.txt");
testbench_read.mipsfpganexys4_read.mipsfpga_sys.mipsfpga_ahb.mipsfpga_ahb_ram_reset.sram_interface_15.TSDN65LPA65536X16M32S_2.MX.load("../rtl_up/initfiles/3_Switches&LEDs/ram_reset_init_high.txt");
testbench_read.mipsfpganexys4_read.mipsfpga_sys.mipsfpga_ahb.mipsfpga_ahb_ram.sram_interface_16.TSDN65LPA65536X16M32S_1.MX.load("../rtl_up/initfiles/3_Switches&LEDs/ram_program_init_low.txt");
testbench_read.mipsfpganexys4_read.mipsfpga_sys.mipsfpga_ahb.mipsfpga_ahb_ram.sram_interface_16.TSDN65LPA65536X16M32S_2.MX.load("../rtl_up/initfiles/3_Switches&LEDs/ram_program_init_high.txt");
end

initial
begin
force testbench_read.mipsfpganexys4_read.sw=16'hfeed;
force testbench_read.mipsfpganexys4_read.btnU=1'b0;
force testbench_read.mipsfpganexys4_read.btnD=1'b1;
force testbench_read.mipsfpganexys4_read.btnL=1'b1;
force testbench_read.mipsfpganexys4_read.btnR=1'b0;
force testbench_read.mipsfpganexys4_read.btnC=1'b1;
end

initial
begin
    btnCpuReset  <= 0;
    repeat (100)  @(posedge clk);
    btnCpuReset  <= 1;
    repeat (1000) @(posedge clk);

end

initial
```

```
begin
    $dumpvars;
    $timeformat (-9, 1, "ns", 10);
end
endmodule
```

Listing 47: Testbench for functional simulation

D Script for "Make"

```
DESIGNPATH = /user/yshi/Abschlussarbeit/MIPSfpga/digital/default

SRS_PATH         = ${DESIGNPATH}


PKGS_RTL_memory_tb = \
${SRS_PATH}/SRAM/tsdn65lpa65536x16m32s_200b/VERILOG/
    tsdn65lpa65536x16m32s_200b_ff1p32v0c.v\
${SRS_PATH}/SRAM/tsdn65lplla8192x16m16s_200a/VERILOG/
    tsdn65lplla8192x16m16s_200a_ff1p32v0c.v\

PKGS_TB = \
${SRS_PATH}/sim/testbench.v \

PKGS_TB_NEXYS = \
${SRS_PATH}/sim/testbenchnexys4.v \

PKGS_TB_NEXYS_READ = \
${SRS_PATH}/sim/testbenchnexys4_read.v \

PKGS_TB_NEXYS_loop= \
${SRS_PATH}/sim/testbenchnexys4loop.v \

PKGS_RTL_syn = \
${SRS_PATH}/rtl_up/dataram_2k2way_xilinx.v\
${SRS_PATH}/rtl_up/d_wsram_2k2way_xilinx.v\
${SRS_PATH}/rtl_up/ejtag_reset.v\
${SRS_PATH}/rtl_up/i_wsram_2k2way_xilinx.v\
${SRS_PATH}/rtl_up/m14k_alu_dsp_stub.v\
${SRS_PATH}/rtl_up/m14k_alu_shft_32bit.v\
${SRS_PATH}/rtl_up/m14k_bistctl.v\
${SRS_PATH}/rtl_up/m14k_biu.v\
${SRS_PATH}/rtl_up/m14k_cache_cmp.v\
${SRS_PATH}/rtl_up/m14k_cache_mux.v\
${SRS_PATH}/rtl_up/m14k_cdmmstub.v\
${SRS_PATH}/rtl_up/m14k_clockandlatch.v\
${SRS_PATH}/rtl_up/m14k_clock_buf.v\
${SRS_PATH}/rtl_up/m14k_clock_nogate.v\
${SRS_PATH}/rtl_up/m14k_clockxnorgate.v\
${SRS_PATH}/rtl_up/m14k_cop1_stub.v\
${SRS_PATH}/rtl_up/m14k_cop2_stub.v\
${SRS_PATH}/rtl_up/m14k_core.v\
${SRS_PATH}/rtl_up/m14k_cp1_stub.v\
${SRS_PATH}/rtl_up/m14k_cp2_stub.v\
${SRS_PATH}/rtl_up/m14k_cpu.v\
${SRS_PATH}/rtl_up/m14k_cpz.v\
${SRS_PATH}/rtl_up/m14k_cpz_antitamper_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_eicoffset_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_guest_srs1.v\
${SRS_PATH}/rtl_up/m14k_cpz_guest_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_pc.v\
${SRS_PATH}/rtl_up/m14k_cpz_pc_top.v\
${SRS_PATH}/rtl_up/m14k_cpz_prid.v\
${SRS_PATH}/rtl_up/m14k_cpz_root_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_sps_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_srs1.v\
${SRS_PATH}/rtl_up/m14k_cpz_watch_stub.v\
${SRS_PATH}/rtl_up/m14k_cscramble_scanio_stub.v\
${SRS_PATH}/rtl_up/m14k_cscramble_stub.v\
${SRS_PATH}/rtl_up/m14k_cscramble_tpl.v\
${SRS_PATH}/rtl_up/m14k_dc.v\
${SRS_PATH}/rtl_up/m14k_dc_bistctl.v\
${SRS_PATH}/rtl_up/m14k_dcc.v\
${SRS_PATH}/rtl_up/m14k_dcc_fb.v\
${SRS_PATH}/rtl_up/m14k_dcc_mb_stub.v\
${SRS_PATH}/rtl_up/m14k_dcc_parity_stub.v\
${SRS_PATH}/rtl_up/m14k_dcc_spmb_stub.v\
${SRS_PATH}/rtl_up/m14k_dcc_spstub.v\
${SRS_PATH}/rtl_up/m14k_dspram_ext_stub.v\
${SRS_PATH}/rtl_up/m14k_edp.v\
${SRS_PATH}/rtl_up/m14k_edp_add_simple.v\
${SRS_PATH}/rtl_up/m14k_edp_buf_misc.v\
${SRS_PATH}/rtl_up/m14k_edp_clz.v\
${SRS_PATH}/rtl_up/m14k_edp_clz_4b.v\
${SRS_PATH}/rtl_up/m14k_edp_clz_16b.v\
${SRS_PATH}/rtl_up/m14k_ejt.v\
${SRS_PATH}/rtl_up/m14k_ejt_and2.v\
${SRS_PATH}/rtl_up/m14k_ejt_area.v\
${SRS_PATH}/rtl_up/m14k_ejt_async_rec.v\
${SRS_PATH}/rtl_up/m14k_ejt_async_snd.v\
${SRS_PATH}/rtl_up/m14k_ejt_brk21.v\
${SRS_PATH}/rtl_up/m14k_ejt_bus32mux2.v\
${SRS_PATH}/rtl_up/m14k_ejt_dbrk.v\
${SRS_PATH}/rtl_up/m14k_ejt_gate.v\
${SRS_PATH}/rtl_up/m14k_ejt_ibrk.v\
${SRS_PATH}/rtl_up/m14k_ejt_mux2.v\
${SRS_PATH}/rtl_up/m14k_ejt_pdttcb_stub.v\
${SRS_PATH}/rtl_up/m14k_ejt_tap.v\
${SRS_PATH}/rtl_up/m14k_ejt_tap_dasamstub.v\
${SRS_PATH}/rtl_up/m14k_ejt_tap_fdcstub.v\
${SRS_PATH}/rtl_up/m14k_ejt_tap_pcsamstub.v\
${SRS_PATH}/rtl_up/m14k_ejt_tck.v\
${SRS_PATH}/rtl_up/m14k_fpuclk1_nogate.v\
${SRS_PATH}/rtl_up/m14k_generic_dataram.v\
${SRS_PATH}/rtl_up/m14k_generic_tagram.v\
${SRS_PATH}/rtl_up/m14k_generic_wsram.v\
${SRS_PATH}/rtl_up/m14k_gf_mux2.v\
${SRS_PATH}/rtl_up/m14k_glue.v\
${SRS_PATH}/rtl_up/m14k_ic.v\
${SRS_PATH}/rtl_up/m14k_ic_bistctl.v\
${SRS_PATH}/rtl_up/m14k_icc.v\
${SRS_PATH}/rtl_up/m14k_icc_mb_stub.v\
${SRS_PATH}/rtl_up/m14k_icc_parity_stub.v\
${SRS_PATH}/rtl_up/m14k_icc_spmb_stub.v\
${SRS_PATH}/rtl_up/m14k_icc_spstub.v\
${SRS_PATH}/rtl_up/m14k_icc_umips_stub.v\
${SRS_PATH}/rtl_up/m14k_ispram_ext_stub.v\
${SRS_PATH}/rtl_up/m14k_mdl.v\
${SRS_PATH}/rtl_up/m14k_mdl_add_simple.v\
${SRS_PATH}/rtl_up/m14k_mdl_ctl.v\
${SRS_PATH}/rtl_up/m14k_mdl_dp.v\
${SRS_PATH}/rtl_up/m14k_mmuc.v\
${SRS_PATH}/rtl_up/m14k_mpc.v\
${SRS_PATH}/rtl_up/m14k_mpc_ctl.v\
${SRS_PATH}/rtl_up/m14k_mpc_dec.v\
${SRS_PATH}/rtl_up/m14k_mpc_exc.v\
${SRS_PATH}/rtl_up/m14k_rf_reg.v\
${SRS_PATH}/rtl_up/m14k_rf_rngc.v\
${SRS_PATH}/rtl_up/m14k_rf_stub.v\
${SRS_PATH}/rtl_up/m14k_siu.v\
${SRS_PATH}/rtl_up/m14k_siu_int_sync.v\
${SRS_PATH}/rtl_up/m14k_spram_top.v\
${SRS_PATH}/rtl_up/m14k_ssram_sp_bw.v\
${SRS_PATH}/rtl_up/m14k_tlb.v\
${SRS_PATH}/rtl_up/m14k_tlb_collector.v\
${SRS_PATH}/rtl_up/m14k_tlb_cpy.v\
${SRS_PATH}/rtl_up/m14k_tlb_ctl.v\
${SRS_PATH}/rtl_up/m14k_tlb_dtlb.v\
${SRS_PATH}/rtl_up/m14k_tlb_itlb.v\
${SRS_PATH}/rtl_up/m14k_tlb_jtlb1entry.v\
${SRS_PATH}/rtl_up/m14k_tlb_jtlb4entries.v\
${SRS_PATH}/rtl_up/m14k_tlb_jtlb16.v\
${SRS_PATH}/rtl_up/m14k_tlb_jtlb16entries.v\
${SRS_PATH}/rtl_up/m14k_tlb_utlb.v\
${SRS_PATH}/rtl_up/m14k_tlb_utlbentry.v\
${SRS_PATH}/rtl_up/m14k_top.v\
${SRS_PATH}/rtl_up/m14k_udi_stub.v\
${SRS_PATH}/rtl_up/mipsfpga_ahb.v\
${SRS_PATH}/rtl_up/mipsfpga_ahb_gpio.v\
${SRS_PATH}/rtl_up/mipsfpga_ahb_ram.v\
${SRS_PATH}/rtl_up/mipsfpga_ahb_ram_reset.v\
${SRS_PATH}/rtl_up/mipsfpga_nexys4.v\
${SRS_PATH}/rtl_up/mipsfpga_sys.v\
${SRS_PATH}/rtl_up/mips_pib_stub.v\
${SRS_PATH}/rtl_up/mvp_cregister.v\
${SRS_PATH}/rtl_up/mvp_cregister_c.v\
${SRS_PATH}/rtl_up/mvp_cregister_ngc.v\
${SRS_PATH}/rtl_up/mvp_cregister_s.v\
${SRS_PATH}/rtl_up/mvp_cregister_wide.v\
${SRS_PATH}/rtl_up/mvp_cregister_wide_tlb.v\
${SRS_PATH}/rtl_up/mvp_cregister_wide_utlb.v\
${SRS_PATH}/rtl_up/mvp_latchn.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_3.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_4.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_5.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_6.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_8.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_9.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_10.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_13.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_24.v\
${SRS_PATH}/rtl_up/mvp_mux2.v\
${SRS_PATH}/rtl_up/mvp_mux4.v\
${SRS_PATH}/rtl_up/mvp_mux8.v\
${SRS_PATH}/rtl_up/mvp_mux16.v\
${SRS_PATH}/rtl_up/mvp_register.v\
${SRS_PATH}/rtl_up/mvp_register_c.v\
${SRS_PATH}/rtl_up/mvp_register_ngc.v\
${SRS_PATH}/rtl_up/mvp_register_s.v\
${SRS_PATH}/rtl_up/mvp_ucregister_wide.v\
${SRS_PATH}/rtl_up/sram_interface.v\
${SRS_PATH}/rtl_up/tagram_2k2way_xilinx.v\
${SRS_PATH}/rtl_up/segment7.v\

#${SRS_PATH}/rtl/clk_wiz_0/clk_wiz_0.v\
#${SRS_PATH}/rtl/clk_wiz_0/clk_wiz_0_clk_wiz.v\
#${SRS_PATH}/rtl/clk_wiz_0/clk_wiz_0_stub.v\



PKGS_RTL_syn_tb= \
${SRS_PATH}/rtl_up/dataram_2k2way_xilinx.v\
${SRS_PATH}/rtl_up/d_wsram_2k2way_xilinx.v\
${SRS_PATH}/rtl_up/ejtag_reset.v\
${SRS_PATH}/rtl_up/i_wsram_2k2way_xilinx.v\
${SRS_PATH}/rtl_up/m14k_alu_dsp_stub.v\
${SRS_PATH}/rtl_up/m14k_alu_shft_32bit.v\
${SRS_PATH}/rtl_up/m14k_bistctl.v\
${SRS_PATH}/rtl_up/m14k_cache_cmp.v\
${SRS_PATH}/rtl_up/m14k_cache_mux.v\
${SRS_PATH}/rtl_up/m14k_cdmmstub.v\
${SRS_PATH}/rtl_up/m14k_clockandlatch.v\
${SRS_PATH}/rtl_up/m14k_clock_buf.v\
${SRS_PATH}/rtl_up/m14k_clock_nogate.v\
${SRS_PATH}/rtl_up/m14k_clockxnorgate.v\
${SRS_PATH}/rtl_up/m14k_cop1_stub.v\
${SRS_PATH}/rtl_up/m14k_cop2_stub.v\
${SRS_PATH}/rtl_up/m14k_cp1_stub.v\
${SRS_PATH}/rtl_up/m14k_cp2_stub.v\
${SRS_PATH}/rtl_up/m14k_cpu.v\
${SRS_PATH}/rtl_up/m14k_cpz.v\
${SRS_PATH}/rtl_up/m14k_cpz_antitamper_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_eicoffset_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_guest_srs1.v\
${SRS_PATH}/rtl_up/m14k_cpz_guest_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_pc.v\
${SRS_PATH}/rtl_up/m14k_cpz_pc_top.v\
${SRS_PATH}/rtl_up/m14k_cpz_prid.v\
${SRS_PATH}/rtl_up/m14k_cpz_root_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_sps_stub.v\
${SRS_PATH}/rtl_up/m14k_cpz_srs1.v\
${SRS_PATH}/rtl_up/m14k_cpz_watch_stub.v\
${SRS_PATH}/rtl_up/m14k_cscramble_scanio_stub.v\
${SRS_PATH}/rtl_up/m14k_cscramble_stub.v\
${SRS_PATH}/rtl_up/m14k_cscramble_tpl.v\
${SRS_PATH}/rtl_up/m14k_dc.v\
${SRS_PATH}/rtl_up/m14k_dc_bistctl.v\
${SRS_PATH}/rtl_up/m14k_dcc.v\
${SRS_PATH}/rtl_up/m14k_dcc_fb.v\
${SRS_PATH}/rtl_up/m14k_dcc_mb_stub.v\
${SRS_PATH}/rtl_up/m14k_dcc_parity_stub.v\
${SRS_PATH}/rtl_up/m14k_dcc_spmb_stub.v\
${SRS_PATH}/rtl_up/m14k_dcc_spstub.v\
${SRS_PATH}/rtl_up/m14k_dspram_ext_stub.v\
${SRS_PATH}/rtl_up/m14k_edp_add_simple.v\
${SRS_PATH}/rtl_up/m14k_edp_buf_misc.v\
${SRS_PATH}/rtl_up/m14k_edp_clz.v\
${SRS_PATH}/rtl_up/m14k_edp_clz_4b.v\
${SRS_PATH}/rtl_up/m14k_edp_clz_16b.v\
${SRS_PATH}/rtl_up/m14k_ejt.v\
${SRS_PATH}/rtl_up/m14k_ejt_and2.v\
${SRS_PATH}/rtl_up/m14k_ejt_area.v\
${SRS_PATH}/rtl_up/m14k_ejt_async_rec.v\
${SRS_PATH}/rtl_up/m14k_ejt_async_snd.v\
${SRS_PATH}/rtl_up/m14k_ejt_bus32mux2.v\
${SRS_PATH}/rtl_up/m14k_ejt_dbrk.v\
${SRS_PATH}/rtl_up/m14k_ejt_gate.v\
${SRS_PATH}/rtl_up/m14k_ejt_ibrk.v\
${SRS_PATH}/rtl_up/m14k_ejt_mux2.v\
${SRS_PATH}/rtl_up/m14k_ejt_pdttcb_stub.v\
${SRS_PATH}/rtl_up/m14k_ejt_tap.v\
${SRS_PATH}/rtl_up/m14k_ejt_tap_dasamstub.v\
${SRS_PATH}/rtl_up/m14k_ejt_tap_fdcstub.v\
${SRS_PATH}/rtl_up/m14k_ejt_tap_pcsamstub.v\
${SRS_PATH}/rtl_up/m14k_ejt_tck.v\
${SRS_PATH}/rtl_up/m14k_fpuclk1_nogate.v\
${SRS_PATH}/rtl_up/m14k_gf_mux2.v\
${SRS_PATH}/rtl_up/m14k_glue.v\
${SRS_PATH}/rtl_up/m14k_ic.v\
${SRS_PATH}/rtl_up/m14k_ic_bistctl.v\
${SRS_PATH}/rtl_up/m14k_icc_mb_stub.v\
${SRS_PATH}/rtl_up/m14k_icc_parity_stub.v\
${SRS_PATH}/rtl_up/m14k_icc_spmb_stub.v\
${SRS_PATH}/rtl_up/m14k_icc_spstub.v\
${SRS_PATH}/rtl_up/m14k_icc_umips_stub.v\
${SRS_PATH}/rtl_up/m14k_ispram_ext_stub.v\
${SRS_PATH}/rtl_up/m14k_mdl.v\
${SRS_PATH}/rtl_up/m14k_mdl_add_simple.v\
${SRS_PATH}/rtl_up/m14k_mdl_ctl.v\
${SRS_PATH}/rtl_up/m14k_mdl_dp.v\
${SRS_PATH}/rtl_up/m14k_mmuc.v\
${SRS_PATH}/rtl_up/m14k_mpc.v\
${SRS_PATH}/rtl_up/m14k_mpc_ctl.v\
${SRS_PATH}/rtl_up/m14k_mpc_dec.v\
${SRS_PATH}/rtl_up/m14k_mpc_exc.v\
${SRS_PATH}/rtl_up/m14k_rf_reg.v\
${SRS_PATH}/rtl_up/m14k_rf_rngc.v\
${SRS_PATH}/rtl_up/m14k_rf_stub.v\
${SRS_PATH}/rtl_up/m14k_siu.v\
${SRS_PATH}/rtl_up/m14k_siu_int_sync.v\
${SRS_PATH}/rtl_up/m14k_spram_top.v\
${SRS_PATH}/rtl_up/m14k_ssram_sp_bw.v\
${SRS_PATH}/rtl_up/m14k_tlb.v\
${SRS_PATH}/rtl_up/m14k_tlb_collector.v\
${SRS_PATH}/rtl_up/m14k_tlb_cpy.v\
${SRS_PATH}/rtl_up/m14k_tlb_ctl.v\
${SRS_PATH}/rtl_up/m14k_tlb_dtlb.v\
${SRS_PATH}/rtl_up/m14k_tlb_itlb.v\
${SRS_PATH}/rtl_up/m14k_tlb_jtlb1entry.v\
${SRS_PATH}/rtl_up/m14k_tlb_jtlb4entries.v\
${SRS_PATH}/rtl_up/m14k_tlb_jtlb16.v\
${SRS_PATH}/rtl_up/m14k_tlb_jtlb16entries.v\
${SRS_PATH}/rtl_up/m14k_tlb_utlb.v\
${SRS_PATH}/rtl_up/m14k_tlb_utlbentry.v\
${SRS_PATH}/rtl_up/m14k_top.v\
${SRS_PATH}/rtl_up/m14k_udi_stub.v\
${SRS_PATH}/rtl_up/mipsfpga_ahb.v\
${SRS_PATH}/rtl_up/mipsfpga_ahb_gpio.v\
${SRS_PATH}/rtl_up/mipsfpga_ahb_ram.v\
${SRS_PATH}/rtl_up/mipsfpga_ahb_ram_reset.v\
${SRS_PATH}/rtl_up/mipsfpga_sys.v\
${SRS_PATH}/rtl_up/mips_pib_stub.v\
${SRS_PATH}/rtl_up/mvp_cregister.v\
${SRS_PATH}/rtl_up/mvp_cregister_c.v\
${SRS_PATH}/rtl_up/mvp_cregister_ngc.v\
${SRS_PATH}/rtl_up/mvp_cregister_s.v\
${SRS_PATH}/rtl_up/mvp_cregister_wide.v\
${SRS_PATH}/rtl_up/mvp_cregister_wide_tlb.v\
${SRS_PATH}/rtl_up/mvp_cregister_wide_utlb.v\
${SRS_PATH}/rtl_up/mvp_latchn.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_3.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_4.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_5.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_6.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_8.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_9.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_10.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_13.v\
${SRS_PATH}/rtl_up/mvp_mux1hot_24.v\
${SRS_PATH}/rtl_up/mvp_mux2.v\
${SRS_PATH}/rtl_up/mvp_mux4.v\
${SRS_PATH}/rtl_up/mvp_mux8.v\
${SRS_PATH}/rtl_up/mvp_mux16.v\
${SRS_PATH}/rtl_up/mvp_register.v\
${SRS_PATH}/rtl_up/mvp_register_c.v\
${SRS_PATH}/rtl_up/mvp_register_ngc.v\
${SRS_PATH}/rtl_up/mvp_register_s.v\
${SRS_PATH}/rtl_up/mvp_ucregister_wide.v\
${SRS_PATH}/rtl_up/sram_interface.v\
${SRS_PATH}/rtl_up/tagram_2k2way_xilinx.v\
${SRS_PATH}/rtl_up/mipsfpga_nexys4.v\
${SRS_PATH}/rtl_up/segment7.v\
${SRS_PATH}/rtl_up/m14k_biu.v\
${SRS_PATH}/rtl_up/m14k_edp.v\
${SRS_PATH}/rtl_up/m14k_ejt_brk21.v\
${SRS_PATH}/rtl_up/m14k_icc.v\
${SRS_PATH}/rtl_up/m14k_core.v\




PKGS_RTL_triplicate = \


PKGS_RTL_do_not_triplicate = \


PKGS_MODEL = \


PKGS_TMR_EL = \


RM = rm -rf

sim_rtl   :
	@echo "Starting␣simulation..."
```

```
         xrun -sv -gui -l ../sim/logs/funciton_run.log -access +
            rwc ${PKGS_RTL_syn_tb} ${PKGS_TB_NEXYS_READ} ${
            PKGS_RTL_memory_tb} -incdir ${SRS_PATH}/rtl_up
            -timescale 1ns/10ps
        #xrun -sv -gui -l ../sim/logs/funciton_run.log -access +
           rwc ${PKGS_RTL_syn_tb} ${PKGS_TB_NEXYS} ${
           PKGS_RTL_memory_tb} -incdir ${SRS_PATH}/rtl_up
           -xverbose -timescale 1ns/1ps


##


sim_gls   :
        @echo "Starting␣simulation..."
        xrun -sv -gui -l ../sim/logs/run_post_pnr_sim.log -access
           +rwc ../output/export.v /eda/kits/TSMC/CERN_C65LP/
           digital/Front_End/verilog/tcbn65lp_200a/tcbn65lp.v ${
           PKGS_TB_NEXYS_READ} ${PKGS_RTL_memory_tb} -timescale 1
           ns/10ps -mess -ntc_level 2 -ntc_verbose -sdf_nopathedge
            -clean -dfile ../sim/dfile.txt -input ../sim/scripts/
           dfile.tcl -clean -sdf_cmd_file ../sim/scripts/
           sdf_typ.cmd -clean -input ../sim/scripts/
           export_saif_shm_typ.tcl

sim_gls_loop   :
        @echo "Starting␣simulation..."
        xrun -sv -gui -l ../sim/logs/run_post_pnr_sim_loop.log
           -access +rwc ../output/export.v /eda/kits/TSMC/
           CERN_C65LP/digital/Front_End/verilog/tcbn65lp_200a/
           tcbn65lp.v ${PKGS_TB_NEXYS_loop} ${PKGS_RTL_memory_tb}
           -timescale 1ns/10ps -mess -ntc_level 2 -ntc_verbose
           -sdf_nopathedge -clean -dfile ../sim/dfile.txt -input
           ../sim/scriptsloop/dfile.tcl -clean -sdf_cmd_file ../
           sim/scriptsloop/sdf_typ.cmd -clean -input ../sim/
           scriptsloop/export_saif_shm_typ.tcl

#
##xminit_log ../sim/logs/init_list.log
#-xminitialize 0 -input ../sim/logs/init_list.log
#-input ../sim/scripts/deposit.tcl
#-dfile ../sim/dfile.txt -input ../sim/scripts/dfile.tcl
#-setenv SHM_RESET_DEFAULTS=TRUE
#-input ../sim/scripts/deposit_new.tcl

```

```
sim_tmr    :
	@echo "Starting␣simulation..."
	xrun -sv -gui -l run.log -access +rwc ${PKGS_MODEL} ${PKGS_RTL_syn} ${PKGS_TB} -timescale 10ps/10ps

try   :
	@echo "Starting␣simulation..."
	irun -sv -c -l run.log -access +rwc ${PKGS_RTL_syn_tb} ${PKGS_TB} ${PKGS_RTL_memory_tb} -incdir ${SRS_PATH}/rtl_up  -timescale 10ps/10ps

clean:
	@echo cleaning old simulation files and libraries...
	@ -$(RM) INCA_libs plib csrc *.out *.err simv* work vlog.opt *.bak *.log *.dat *.txt .simvision ncsim* *.vpd transcript \
	waves.shm *.wlf mylib lib DVEfiles ucli.key irun.key modelsim.ini *.vstf .restart* urgReport cov_work *.so vc_hdrs.h
	irun -clean
	@echo done.

triplicate:
	@echo "Starting␣triplication..."
	tmrg --no-common-definitions ${PKGS_TMR_EL} ${PKGS_RTL_triplicate}
	@mv ./*.*v ../tmr/
	@cp ${PKGS_RTL_do_not_triplicate} ./../tmr/
	touch ./../tmr/TMRdef..v \
	@echo "\`define␣TMR" > ./../tmr/TMRdef..v \
	@echo "Done!"

checktmr:
	@echo "Checking␣triplication␣result..."
	CheckTMR ${SRS_PATH}/rtl/ ${SRS_PATH}/tmr/

.EXPORT_ALL_VARIABLES:

export designpath=${DESIGNPATH}
export PKGS_RTL_syn;

syn:
	@echo "Starting␣Synthesis..."
	#genus -gui -legacy_gui
	genus -files ../syn/scripts/run_all.tcl
```

```

genus:
        @echo "Starting␣Synthesis␣Tool..."
        genus -gui

pnr:
        @echo "Starting␣Place␣and␣Route..."
        innovus -stylus -no_gui -files ../pnr/scripts/0_run_all.tcl

innovus:
        @echo "Starting␣Place␣and␣Route..."
        innovus -stylus
power:
        @echo "Starting␣Voltus␣Power␣Simulation..."
        voltus -vtsxl -stylus -files ${DESIGNPATH}/power/scripts/0_run_all.tcl
powerloop:
        @echo "Starting␣Voltus␣Power␣Simulation..."
        voltus -vtsxl -stylus -files ${DESIGNPATH}/power/scriptsloop/0_run_all.tcl
```

Listing 48: Script Make

Statement of Authorship

I hereby declare that I am the sole author of this bachelor thesis / master thesis (please select) and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

(Place, Date)	(Signature)