Kai Pokrandt
System Synthesis

Final Project: Microcontroller

Project Objective:

The goal of this project was to design, synthesize, and simulate a small 16-bit microcontroller built from modular RTL components and a shared internal bus.

The microcontroller has the following high-level behavioral goals:

- It fetches and executes a small program stored in an external 16-bit word-addressable memory.
- It supports a compact instruction set including register-register ALU ops, immediate ALU ops, register moves, memory loading and storing, and basic input/output through an output port P0 and an input port P1.
- It uses a single shared 16-bit internal bus and central control unit implemented with numerous finite state machines.
- It must be synthesizable to the SAED90nm library we are given, and able to be verified post-synthesis with gate-level timing simulation.

The specific demonstration program used to validate the design performs a sequence of mathematical and logical operations on register R1 and R3, then uses a STORE and LOAD from R3 to move a value through memory and then drive it onto the external port P0. This gives a full test of instruction fetching, decoding, ALU, register files, memory, and the P0 output interfaces.

The testbench program for my design, as depicted by the given document:

- MOVI R1, #14
- ADDI R1, #5
- MOV R3, R1
- SUBI R3, #2
- XOR R3, R1
- NOT R3
- STORE R1, (R3)
- LOAD (R3), P0

Functionally, this loads the constant 14 into R1, then adds immediate 5 to R1 (0x0013). It then copies R1 into R3, then subtracts immediate 2 from R3. Then, it XORs R3 with R1, then inverts R3 via NOT. Finally, it uses R3 as an address to STORE R1 into memory, then uses R3 again as an address to LOAD that memory location into the output port P0. Correct behavior through the entire sequence ideally shows a final value of 0x0013 in the final P0 output.

Kai Pokrandt
System Synthesis

Design Methodology:

My design strategy was entirely modular. I have trouble keeping things organized when everything isn't contained in its own file. Each block has a defined interface and responsibility, and all blocks communicate only through the single shared 16-bit bus with a small number of control signals. This modularity made synthesis extremely irritating initially, but once I got my sdf annotation script working, it sped up tremendously.

Program Counter (kaipokrandt_PC.v)

The program counter is a 16-bit register with three control signals: reset, increment, and enable. When enable is high, the counter's contents are driven onto the internal bus. When increment is asserted on a clock edge, the counter increments itself by 1. On a reset, the counter clears to zero so that fetching begins at address 0 again. The counter is a simple synchronous register with an internal adder for PC+1. Exposing the PC value on the bus instead of directly into the memory address register allows for reuse of the bus and keeps everything spiffy.

Memory Address Register (kaipokrandt_MAR.v)

The MAR is a 16-bit register that captures an address from the bus when load is asserted. Its output is hard wired to the external memory address mem_addr. It is synchronous to the clock and cleared when a reset occurs. The idea here was to keep the MAR as a simple buffer between the internal bus and the external memory. This separates the timing of the bus from the memory's internal enable logic.

Memory Data Register (kaipokrandt_MDR.v)

The MDR is a more complex interface than MAR because it handles both read and write ops. bus_in/out connects to the bus, obviously. Mem_dout/din connects to the external memory data ports. Separate load enables exist for loading from the bus and loading from memory. Enable_bus controls the MDR's ability to drive the bus. I treat the MDR as the only gateway between the internal data path and external memory. All instruction fetches and load/stores pass through this MDR. At a base level, it is just a register with an added tri-state bus output, and a separate register that holds mem_din for storing.

Instruction Register and Decoder (kaipokrandt_IR/ID.v)

The instruction register takes a 16-bit word from the bus when load is given. It splits the word into the three fields needed, being the opcode (4-bit), param1 (6-bit), and param2 (6-bit). The instruction decoder is entirely combinational and takes ir_out as input. The decoder classifies the opcode into 6 possible classes: dec_alu_reg, dec_alu_imm, dec_load, dec_store, dec_mov, dec_movi. It also outputs the 4-bit ALU operation. It calls a flag, uses_imm, to indicate whether or not param2 is an immediate value. This split up word is used by the microcontroller. The design for both modules is to keep IR sequential and ID combinational. This way, it ensures that when an instruction is fetched, it's decoded data remains stable for the whole EXEC state, which allows our FSMs (ALU, MEM, MOV, MOVI) to use it for multiple cycles without the instruction changing.

ALU Core (kaipokrandt_ALU.v)

The ALU is a multi-cycle unit using internal registers. It contains two op latches, in1_ld and in2_ld, an output register, out_ld, a bus output enable, alu_out_en, and a 4-bit opcode select, alu_op. The ALU operations include ADD, SUB, OR, XOR, XNOR, NOT, and ADDI/SUBI. These FSMs schedule

the bus transfers so operands are moved one at a time from the register file, onto the bus, and then into the ALU input latches. On the next cycle, the ALU computes the result and, if out_ld and alu_out_en are high, it will drive the result of the operation on the bus for the destination register to absorb. This is a time-multiplexed ALU. The bus carries one operand at a time. This makes the design simple in waveforms which helps immensely with debugging.

Register File (kaipokrandt_regtristate.v, R0->R3, P0)

The general purpose registers are implemented as 4 individual regtristates. Each register has a synchronous load input, which takes the bus value on a clock edge. It also has an enable input which allows that specific register's output to be taken into account for driving the bus, as well as having separate bus_in and bus_out ports. In the top level, the enables and loads for R0-R3 are decoded based on the 'destinationr' and 'sourcer' fields and control signals from all the FSMs (reg_dst_reg_ld, reg_src_reg_en, mem_dst_reg_ld, etc.). Only one of the R0-R3 registers is allowed to drive the bus at a time, and only when required by the operation being done. Port P0 is used as a fifth register related element. It has two roles. Internally, it has a register that loads the bus when it's asserted. Externally, it drives p0_ext_out when extout is high.

Memory (kaipokrandt_bmem.v)

The memory is a behavioral module. 16-bit address, 16-bit data in, 16-bit data out. It has a control enable and clock, a readwrite where 1=read/0=write. It also has an MFC (memory function complete) flag that raises after an operation. The storage is a block of 64k words. The initial sets all locations to 0, then sets address 0 and 1 to arbitrary values that are overwritten by the top level testbench. On the rising edge of enable, if readwrite is 1, dataout is assigned from the memory address, if not, memory address is assigned from data in. On the falling edge of enable, MFC is turned off. Methodologically, memory is separated from the synthesizable core of the rest of the controller. The memory is never EVER passed through Design Vision/ It is a purely simulation model that is initialized in the testbench.

Control FSMs

Control logic is split into numerous FSM modules. FETCH, ALU REG, ALU IMM, MEM, MOV, MOVI, CPU FETCH, CPU EXEC.

Fetch FSM (kaipokrandt_fsm_FETCH.v)

This is responsible for taking in instructions. It enables the program counter on the bus so the memory address register can load the instruction address. It asserts mem_EN and mem_RW for an instruction read. It can load MDR from memory when the MFC is high. It drives MDR on the bus so the instruction register can load the instruction, then finally, increments the program counter. It outputs fetch_busy and fetch_done so the top level CPU knows when to transition from FETCH to EXEC

ALU Register to Register FSM (kaipokrandt_fsm_REG.v)

This FSM starts when dec_alu_reg is active. In order, it enables a source register onto the bus to latch into the ALU input 1 and 2, then controls the alu_op, alu_out_ld, and alu_out_en. It then generates dst_reg_en and dst_reg_ld to write the result.

ALU Immediate FSM (kaipokrandt_fsm_IMM.v)

This is very similar to the above FSM, but one operand, param2, is an immediate value placed on the bus via imm_val. It asserts imm_to_bus_en to drive onto the bus.

Memory FSM (kaipokrandt_fsm_MEM.v)

This FSM handles LOAD and STORE. For LOAD, it takes memory address register load, starts a read, waits for memory function complete, then loads memory data register from memory, then finally drives MDR onto the bus and sets dst_reg_ld, or p0_load if the destination is P0.

MOV FSM (kaipokrandt_fsm_MOV.v)

This is a simple FSM used when dec_mov is the opcode. It enables the source register, waits a cycle, then sets the destination register load to high.

MOVI FSM (kaipokrandt_fsm_MOVI.v)

This handles the register moves of immediate data into a register. It sets imm_to_bus_en to high so the low 6 bits of the instruction appear on the bus, then sets the destination register load to high.

CPU FETCH/EXEC controller (kaipokrandt_toplvl.v)

This FSM selects ONE of the above FSMs to start when leaving FETCH, based on the instruction decoder outputs. It holds the CPU in EXEC until the FSM's done signal is set to high, then returns to FETCHing for the next instruction.

Top Level Microcontroller (kaipokrandt_toplvl.v)

This top level module connects all the components in the microcontroller and sets up the single internal bus. It is responsible for instantiating PC, MAR, MDR, IR, ID, ALU, R0-R3, Port1, FETCH FSM, all EXEC FSMs, and wires up the external bmem and port0. It generates all register loads and enables based on destination and source register fields and the respective FSM control signals. At the end, it multiplexes the shared bus. It uses a priority-encoded block that selects which source drives bus_r, then assigns bus = bus_r. The bus priority is PC > IMM > MDR > ALU > R0..3 > P0/P1.

Instruction Set/Machine Code:

The instruction word is 16 bits wide and split into 3 sections. First 4 bits are the opcode, next 6 bits are 'param1' which is usually the destination register index. The last 6 bits, 'param2' is either a source register index or a 6-bit immediate value. This instruction set encompasses register-register ALU operations, immediate ALU operations, load and store instructions, move and move-immediate instructions, and finally port operations.

- '0001' (1) = ADDI : add immediate
- '0011' (3) = SUBI : subtract immediate
- '0100' (4) = NOT : bit inverter
- '0111' (7) = XOR : exclusive OR reg to reg.
- '1001' (9) = LOAD : load from memory (regsource) into dest register or P0
- '1010' (A) = STORE : store from dest register into memory at specific address
- '1011' (B) = MOV : move from reg to reg

- ‘1100’ [C] = MOVI : move immediate value into dest register.

In the demonstration testbench:

- MOVI R1, #14 = 0xC04E, opcode C, dest = 1, imm = 14.
- ADDI R1, #5 = 0x1045, opcode 1, dest = 1, imm = 5.
- MOV R3, R1 = 0xB0C1, opcode B, dest = 3, src = 1.
- SUBI R3, #2 = 0x30C2, opcode 3, dest = 3, imm = 2.
- XOR R3, R1 = 0x70C1, opcode 7, dest = 3, src = 1.
- NOT R3 = 0x40C0, opcode 4, dest = 3, src field unused.
- STORE R1, (R3) = 0xA043, opcode A, dest = 1, param2 = 3 (R3 as address)
- LOAD (R3), P0 = 0x9103, opcode 9, dest index = 4 (P0), param2 = 3 (R3 as address)

The ID module maps these opcodes into the high-level control signals. Opcodes 1 and 3 assert dec_alu_imm and choose the appropriate alu_op for addition or subtraction. Opcode 7 asserts dec_alu_reg with alu_op = XOR. Opcodes 9 and 10 assert dec_load and dec_store. Opcode 11 asserts dec_mov and opcode 12 asserts dec_movi.

Assumptions:

Several assumptions underlie this design. First, all registers, including PC, MAR, MDR, IR, and the general purpose registers, are synchronous and update only on the rising edge of the global clk. The only exceptions to this rule are the MFC and some memory behavior.

Second, reset is active-low and asynchronous. When reset is de-asserted (0), all key registers are cleared, CPU state is forced to FETCH, and no FSM is marked as busy. The ensures a clean starting point at time 0.

Third, only one logical source is allowed to drive the internal bus at a time. In RTL, this is enforced by the priority encoder in the bus_r assignment.

Fourth, for simplicity, immediate values are zero-extended rather than sign-extended. This works fine for my given test sequence.

Fifth, memory is assumed to complete a read or write within a single clock pulse. Initially, i tried only using the enable pulse, but stuff was breaking.

Sixth, the CPU always completes and instructions EXEC phase before fetching the next, so values are always stable in registers by the time they are needed again.

Finally, there is no explicit stop the uC button. The testbench just runs for 20000ns as per my request to make sure I see the whole sequence.

Kai Pokrandt
System Synthesis

Synthesis Process:

The synthesis process used the SAED90nm standard cell library and it was exactly like every other synthesized project we've done. We synthesize all parts of the microcontroller except toplvl.v, bmem.v, and the top level testbench. Everything else is synthesized under the same timing constraints for simplicity. This resulted in a positive slack above 15 for all synthesized components. I just used the default constraints given in the original Design Vision tutorial, 30ns period, .14ns uncertainty, 2.0ns input delay and .5ns output delay. We start by setting up the link, target, and symbol libraries given by the tutorial. We then read in all the synthesizable files (all but those listed before), then run constraints, compile, report timings, and create the respective syn.v and .sdf files for all components (again, all but the 3 listed before). Once that is done, we can take all those files back to ModelSim for simulation.

Simulation Method:

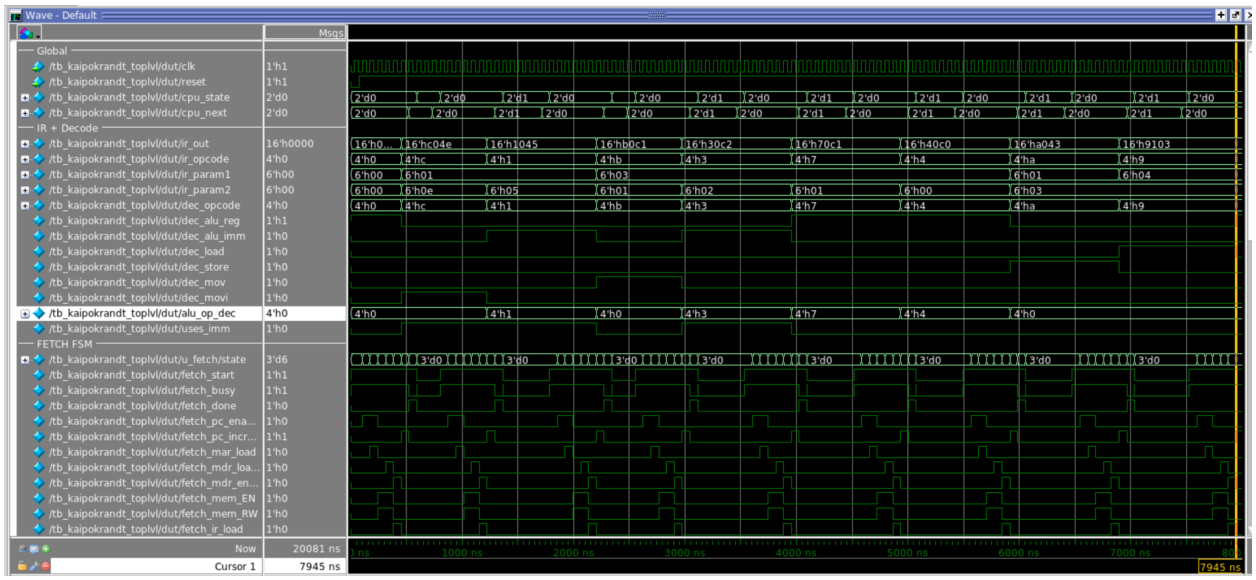
Simulation begins with the RTL functional simulation, then ends with the post-synthesis gate-level simulation. Respective waveforms will be shown (and there are a LOT) in the results section. The testbench synthesizes the top-level microcontroller as dut. The memory, mem, is connected to dut respectively. The testbench preloads the program into the memory via mem.mem[..] in the initial block. This ensures that when the CPU starts at address 0, the correct instructions are present. We create the work library for RTL and work_syn for gate level simulations. The RTL is easy, just vlog all respective files in and vsim the testbench. The gate-level simulation gets tricky. Since we synthesized all files except the 3, bmem, toplvl, and testbench, we have .sdf files that we need to annotate for all syn.v files. This sdf annotation ended up being a massive headache. I eventually typed all of the .sdf annotations out in a .txt file so I could copy-paste them for the numerous debugging runs I had to do. To run the synthesized simulation, you read in the saed90nm library, the syn.v files, then load in the non synthesized memory, toplevel, and toplevel testbench.

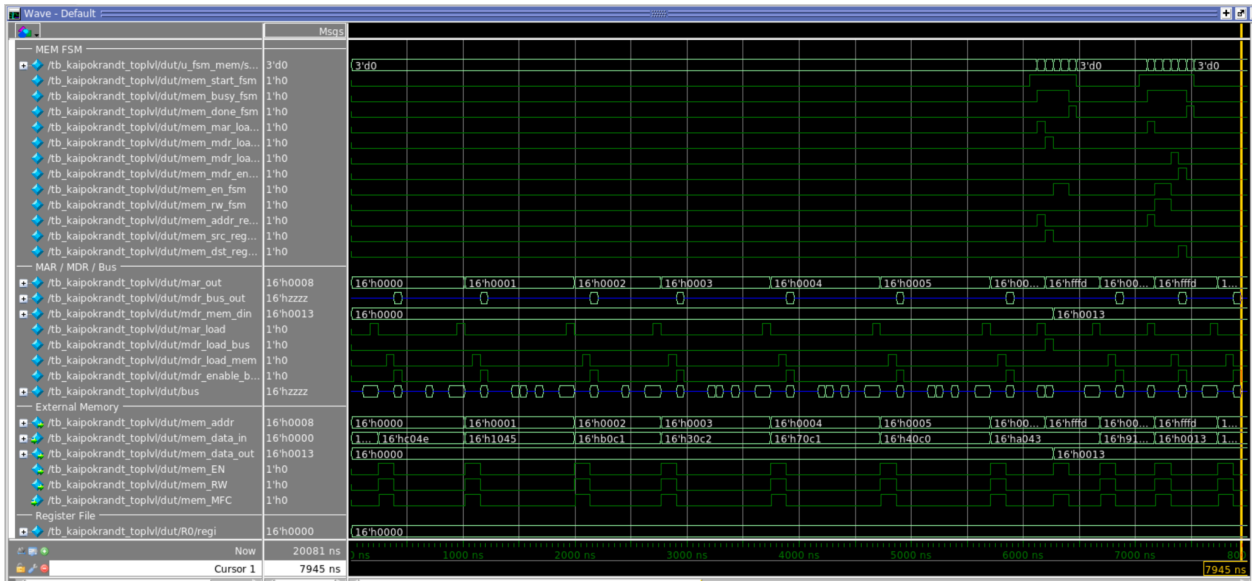
(vsim -novopt tb_kaipokrandt_toplvl -sdfnoerror -sdfnowarn -sdfmax
/tb_kaipokrandt_toplvl/dut/u_pc=../syn/kaipokrandt_PC_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_mar=../syn/kaipokrandt_MAR_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_mdr=../syn/kaipokrandt_MDR_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_ir=../syn/kaipokrandt_IR_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_id=../syn/kaipokrandt_ID_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_alu=../syn/kaipokrandt_alu_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/R0=../syn/kaipokrandt_regtristate_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/R1=../syn/kaipokrandt_regtristate_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/R2=../syn/kaipokrandt_regtristate_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/R3=../syn/kaipokrandt_regtristate_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_p0=../syn/kaipokrandt_port0.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_p1=../syn/kaipokrandt_port1.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_fetch=../syn/kaipokrandt_fsm_FETCH_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_fsm_reg=../syn/kaipokrandt_fsm_REG_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_fsm_imm=../syn/kaipokrandt_fsm_IMM_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_fsm_mem=../syn/kaipokrandt_fsm_MEM_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_fsm_mov=../syn/kaipokrandt_fsm_MOV_sdf.sdf -sdfmax
/tb_kaipokrandt_toplvl/dut/u_fsm_movi=../syn/kaipokrandt_fsm_MOVI_sdf.sdf)

Kai Pokrandt
System Synthesis

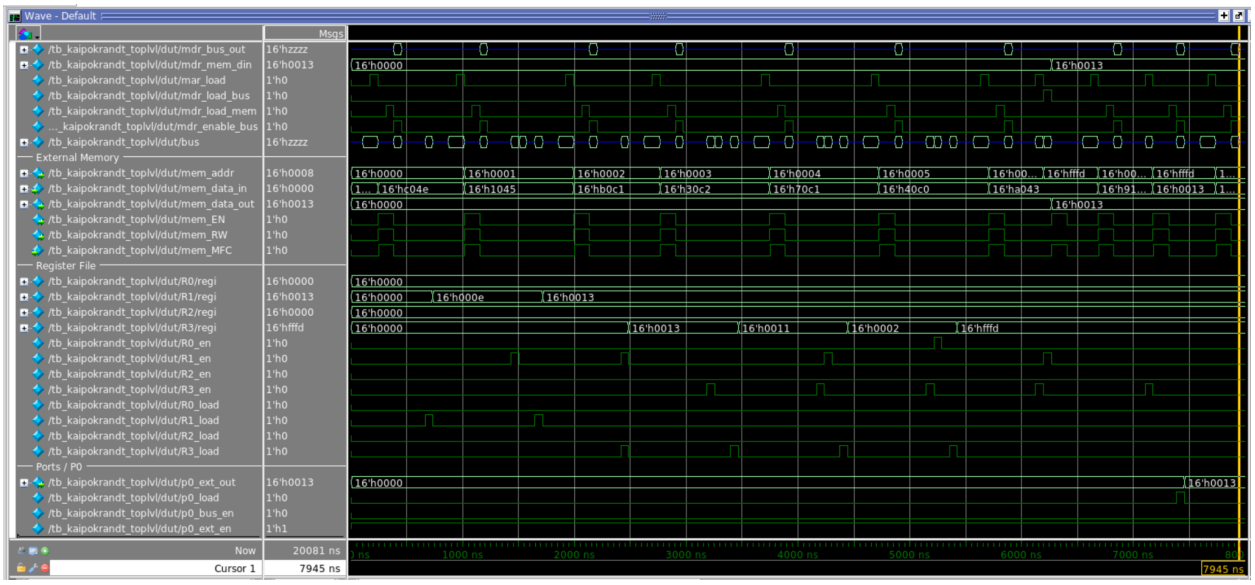
Results:

First, RTL waveforms. The synthesized waveforms are annotated.



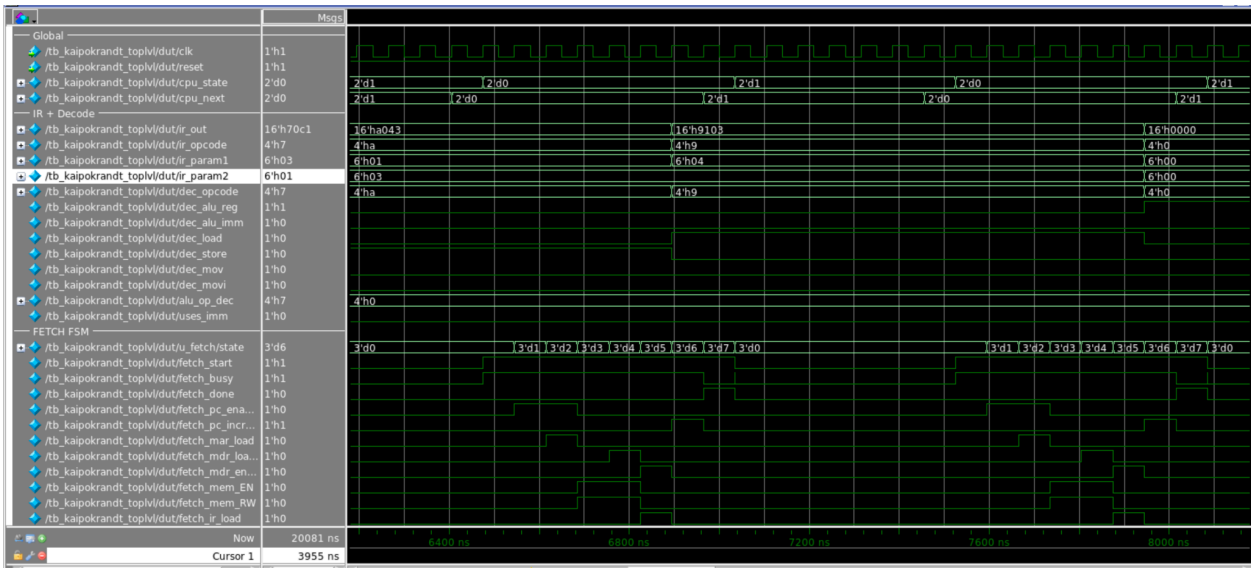


Kai Pokrandt
System Synthesis

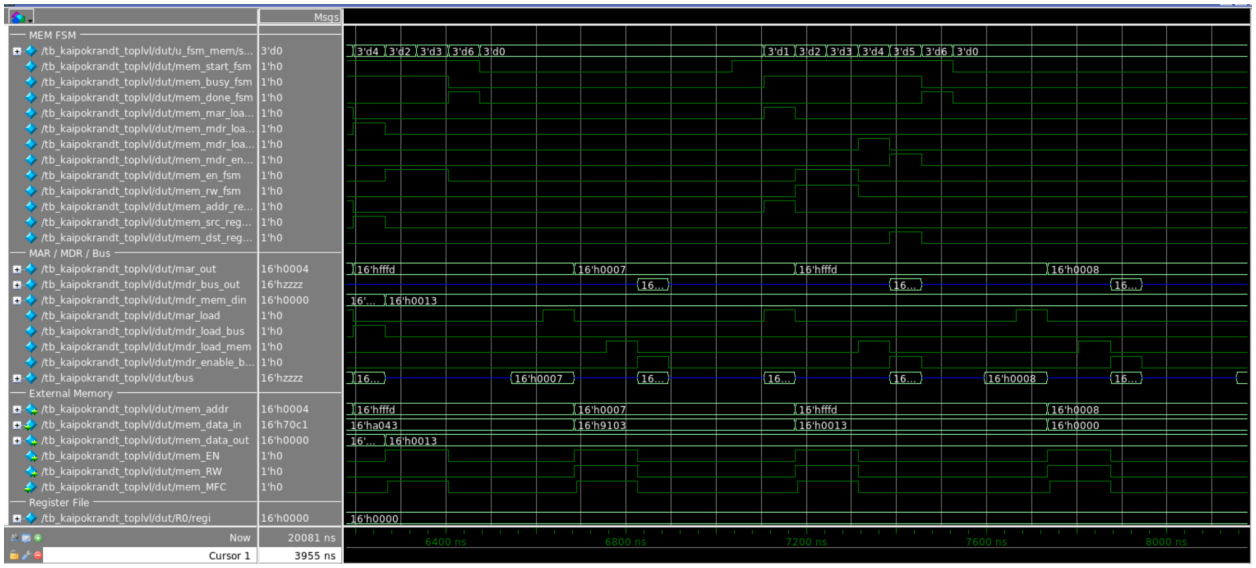


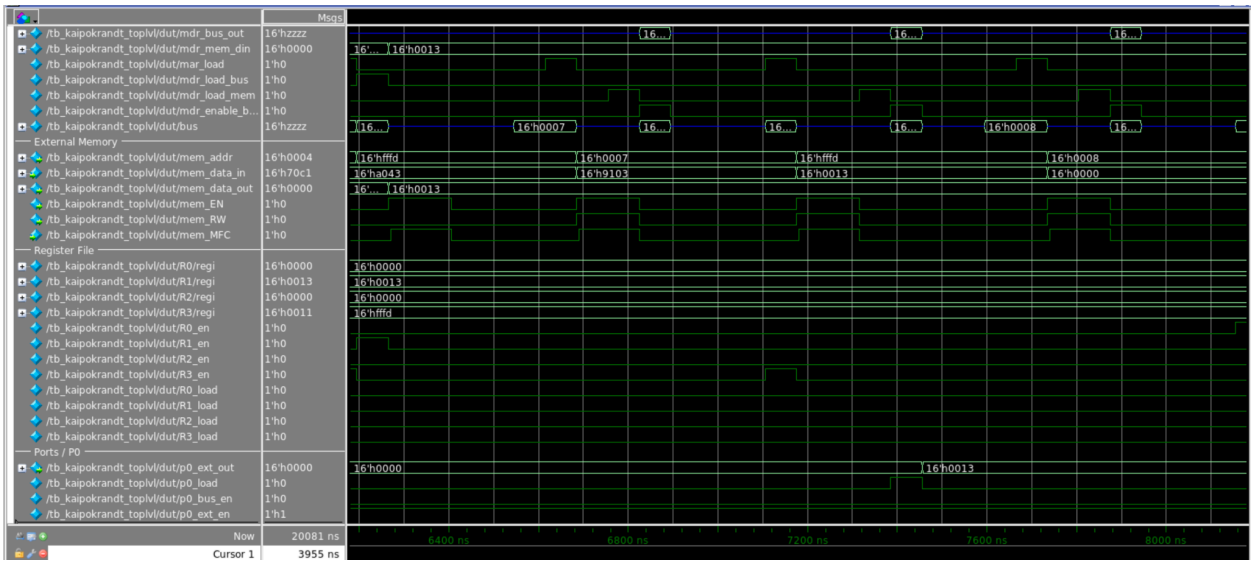
This is zoomed out, showing the whole sequence and the proper final P0 output = 16'h0013, although it's hard to track values this zoomed out. This is specifically to show the ideal output at the end of the whole instruction sequence.

Here is the RTL zoomed in, specifically at the end, top-down in the same time range:



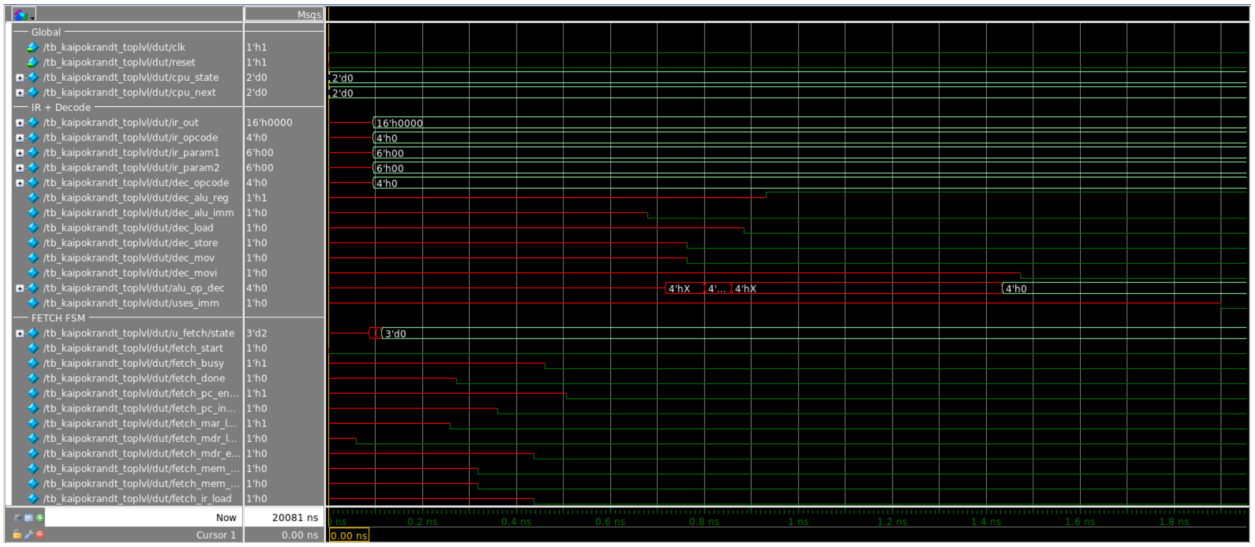
Kai Pokrandt
System Synthesis

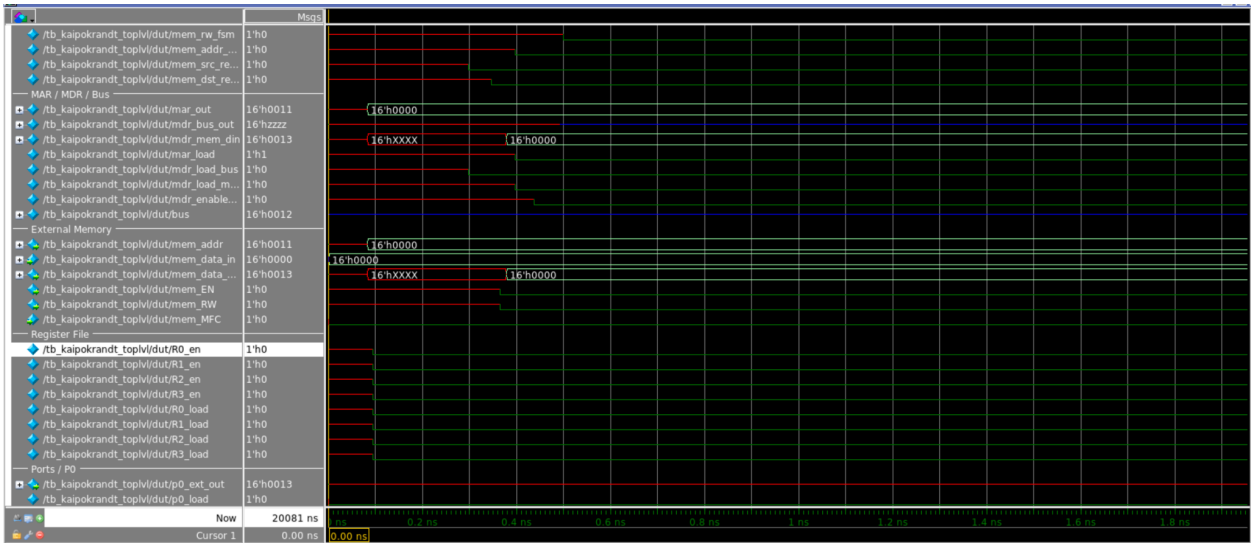




Kai Pokrandt
System Synthesis

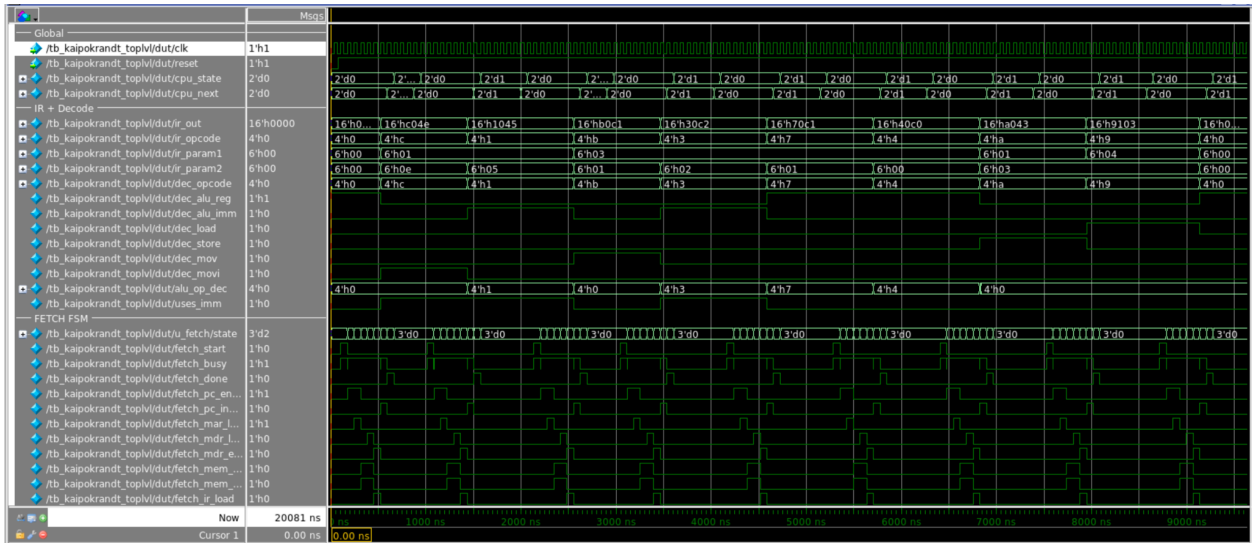
Here are the synthesized waveforms showing the initial delays:

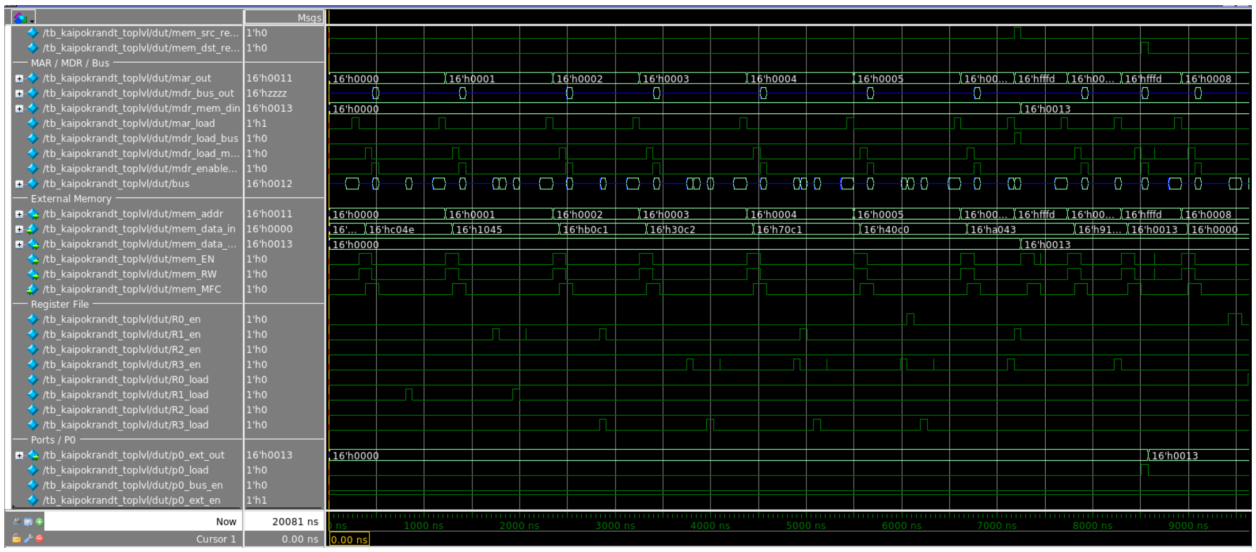




Kai Pokrandt
System Synthesis

Here is the whole (zoomed out) synthesized waveforms:

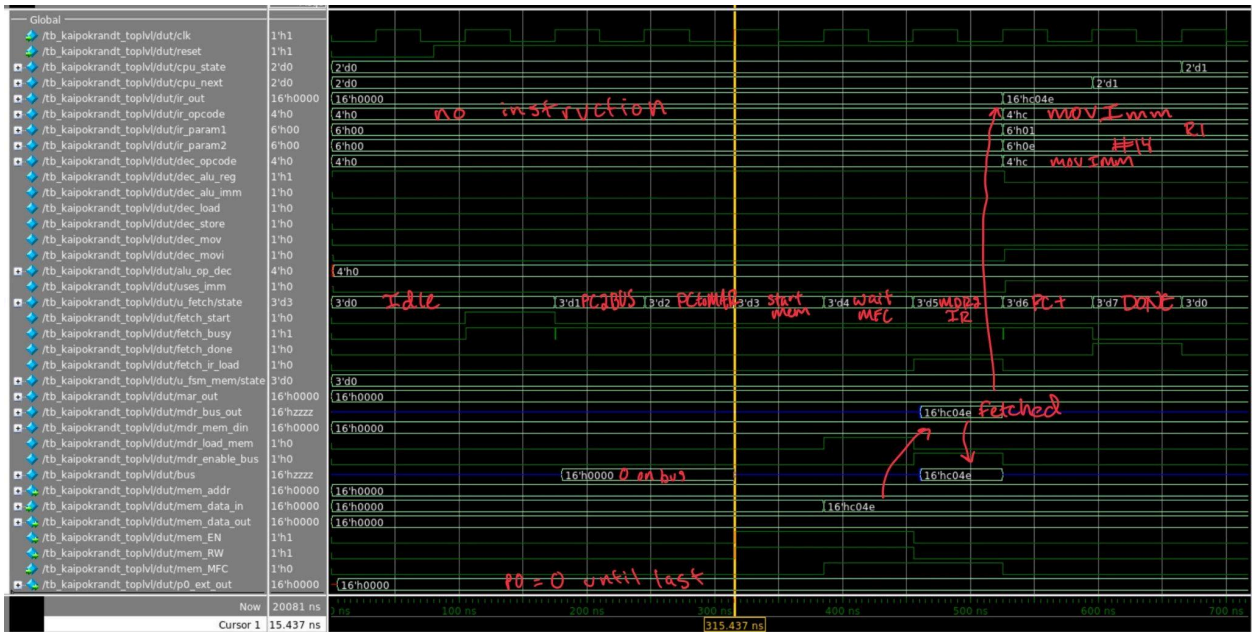




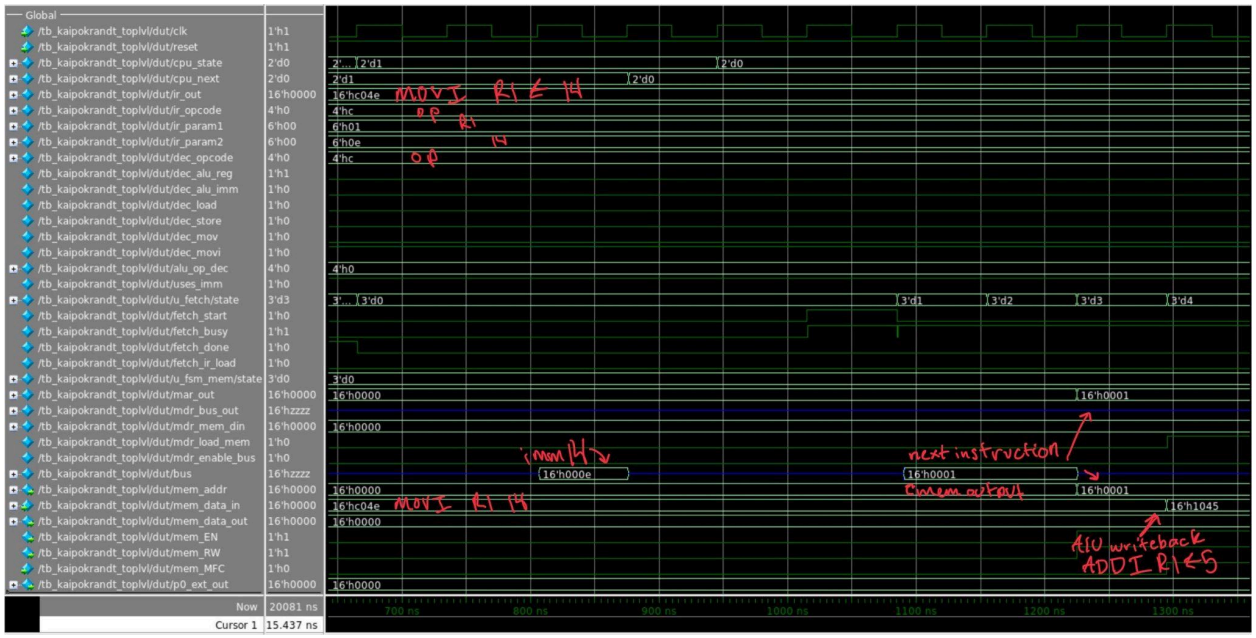
These screenshots are again, to show proper 16’h0013 output in our synthesized waveform as well.

Kai Pokrandt
System Synthesis

Now here is the annotated synthesized waveform showing how the data is transferring to the best of my ability. It shows all bus values throughout the simulation.

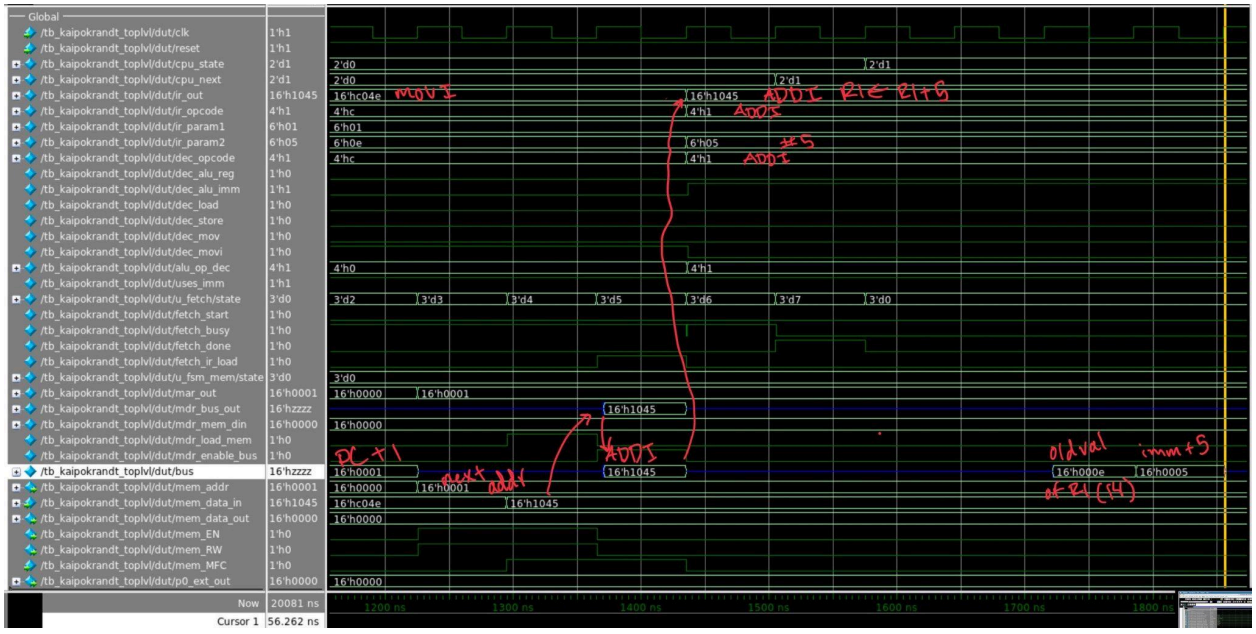


Initially the CPU is in FETCH with PC=0. The bus carries 0x0000, representing the address of the first instruction. The value comes from the PC, enabled by the fetch FSm, and is loaded into the MAR. Shortly after, the bus carries 0xC04E, the instruction for MOVI R1,#14, returned by memory through the MDR and driven onto the bus so that IR can load it. This shows the correct fetch path (PC -> MAR - memory - MDR - IR) is working smoothly.

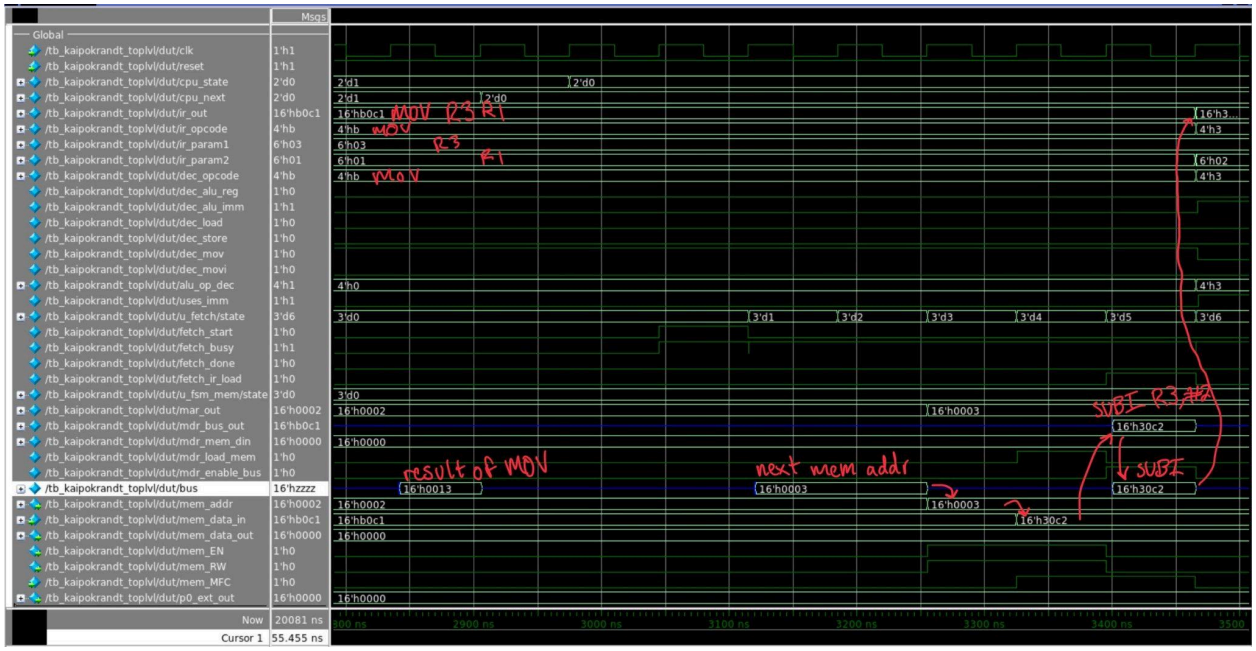


After the MOVI instruction is in IR, the CPU transitions to EXEC. The bus shows 0x000E, the zero-extended immediate value #14. This value is driven by immediate logic, and is captured by R1. Later in this window, the bus carries 0x0001, the incremented program counter value, as PC loads the next address.

Kai Pokrandt
System Synthesis

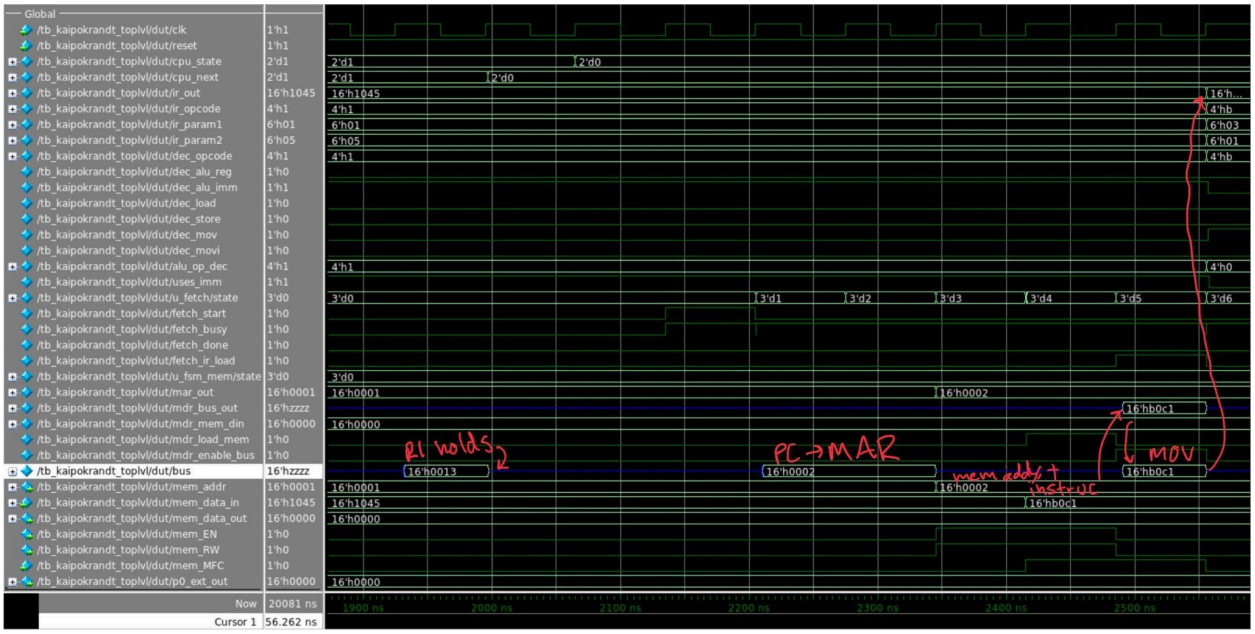


As simulation progresses, the next FETCH brings in the ADDI instruction. The bus again carries the address 0x0001 to MAR, followed by 0x1045, the instruction word for ADDI R1,#5. During this EXEC phase, the bus shows the operand movements required by the ALU. First R1's previous value 0x000E, then the immediate 0x0005, and eventually the ALU results 0x0013 (19) being written to R1. (14+5=19)

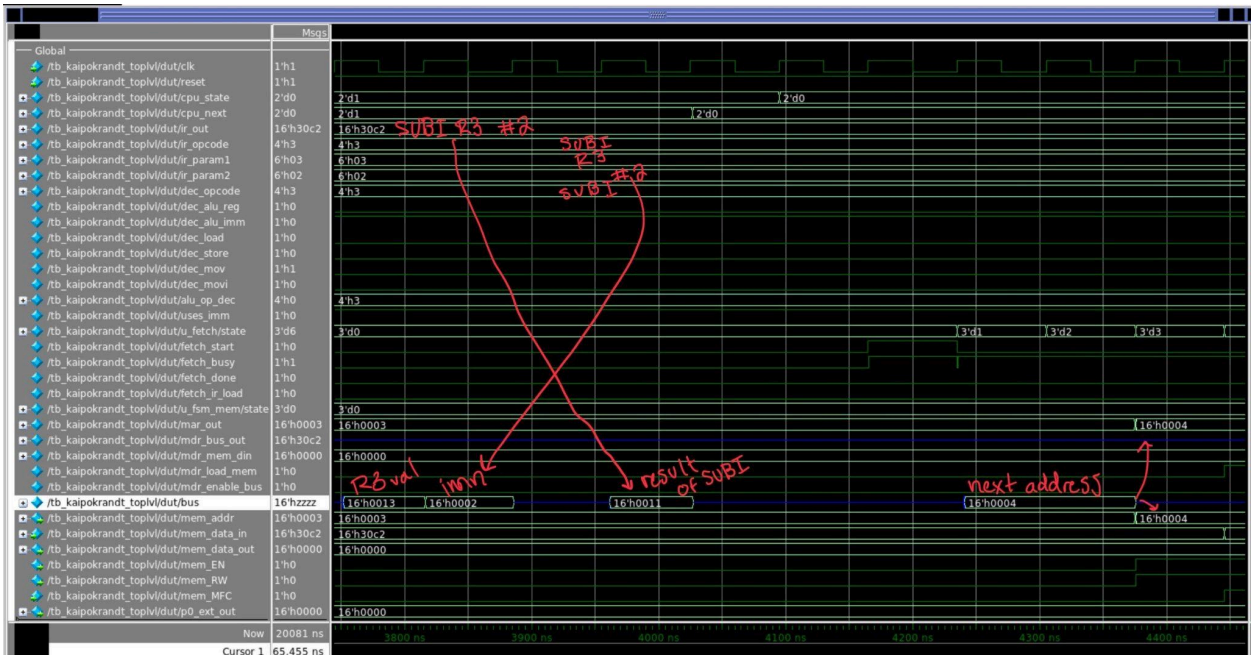


For the MOV R3,R1 instruction, the waveform shows the bus holding 0xB0C1 during FETCH, and later as 0x0013 as R1 is enabled onto the bus and R3 loads the same value.

Kai Pokrandt
System Synthesis

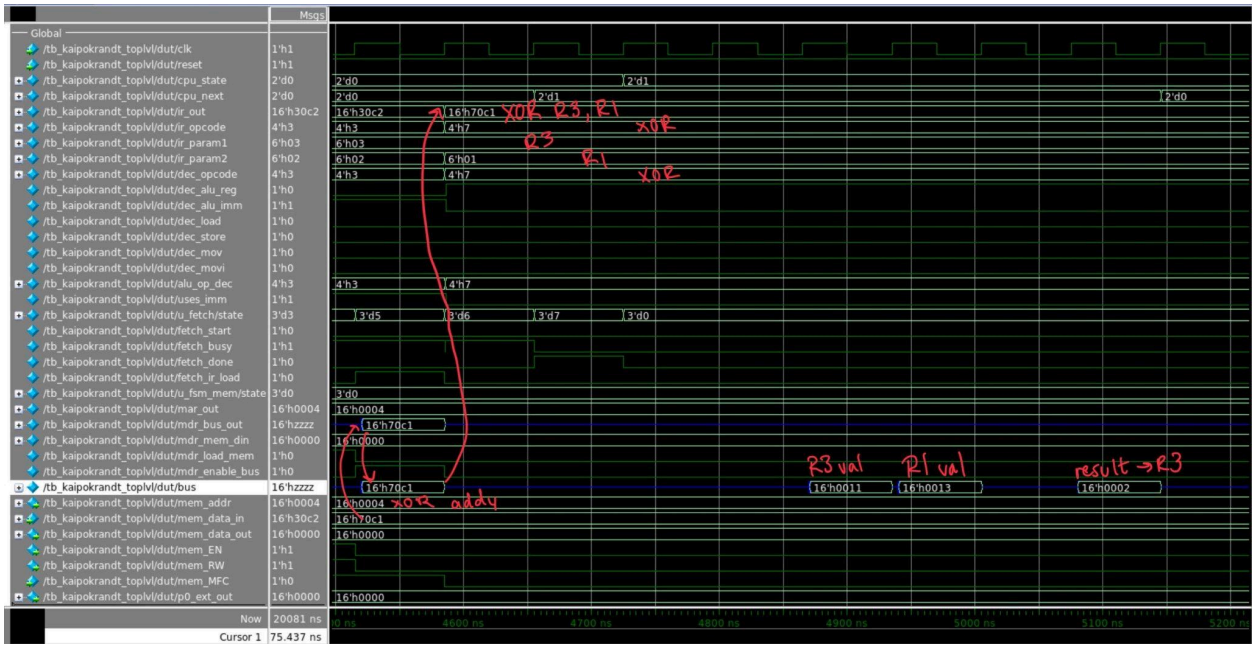


Continued waveform of MOV R3,R1 ^

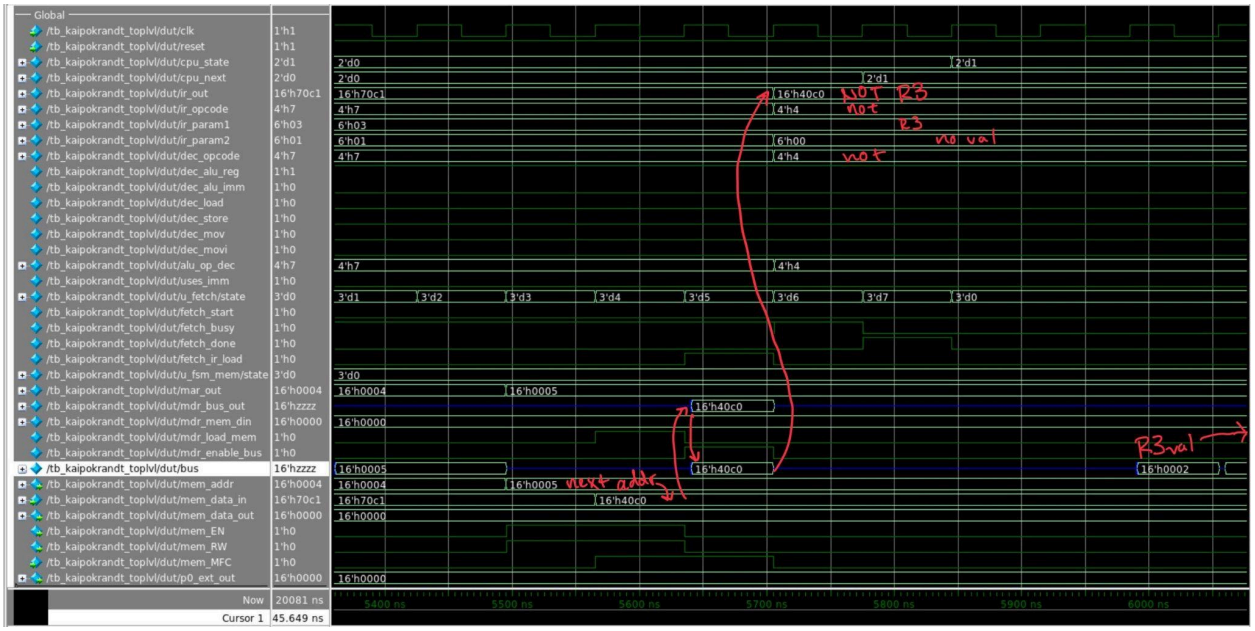


During SUBI R3, #2, the bus carries 0x30C2 as the instruction is fetched. In the EXEC window, the bus shows R3's old value and the immediate 2 being sent to the ALU, followed by 0x0011, confirming the 0x0013 - 0x0002 behavior.

Kai Pokrandt
System Synthesis

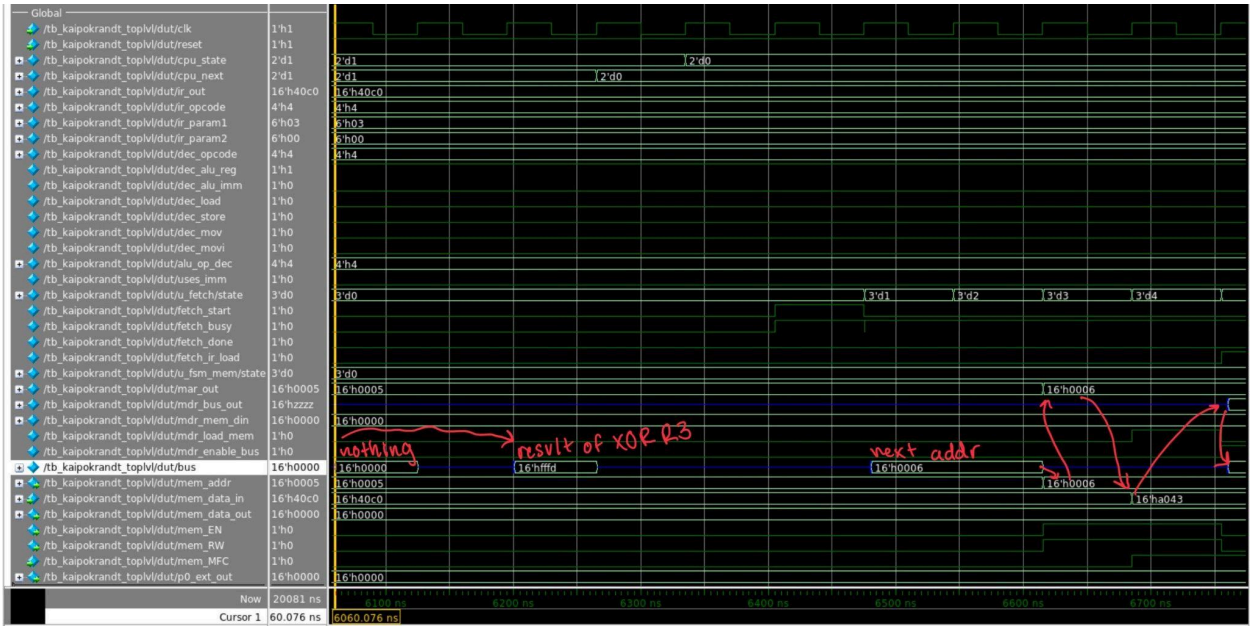


The XOR R3,R1 instruction shows that after 0x70C1 appears on the bus as the fetched instruction, the bus alternates between 0x0011 (R3) and 0x0013 (R1) as operands, then shows 0x0002, the proper XOR result.

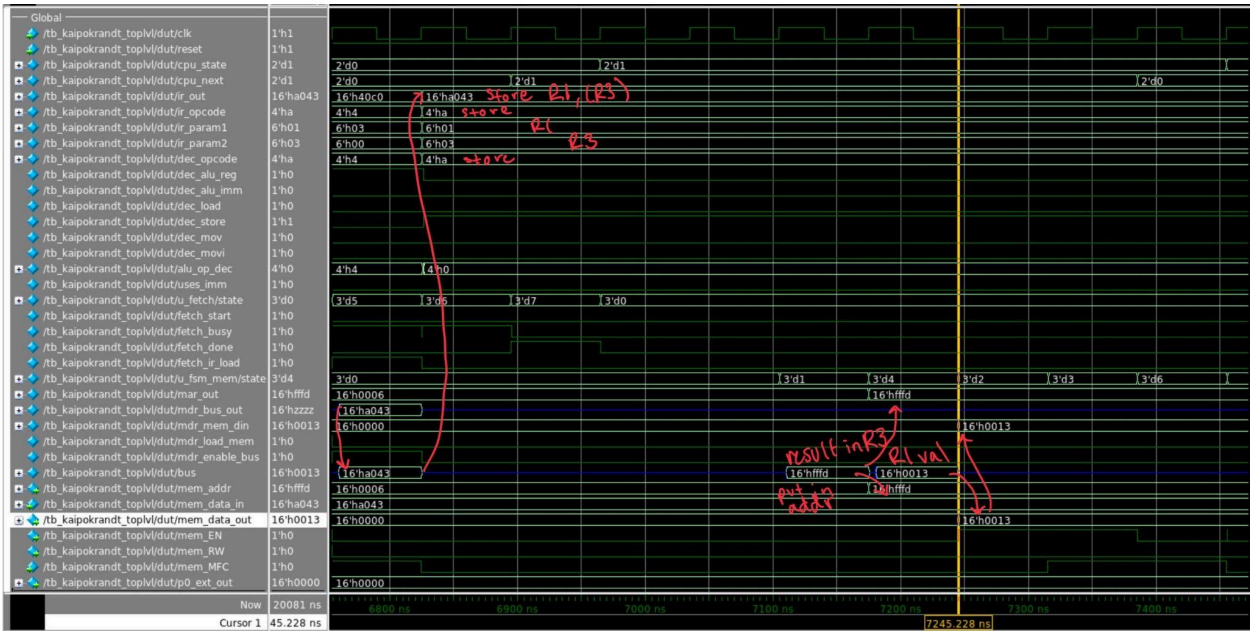


Now we have the NOT R3. The bus carries 0x40C0 as the instruction and later shows 0xFFFD, which is the bitwise complement of 0x0002. This confirms the NOT operation was carried out successfully. R3 contains 0x0002, and after NOT, it holds ~0x0002 = 0xFFFD.

Kai Pokrandt
System Synthesis

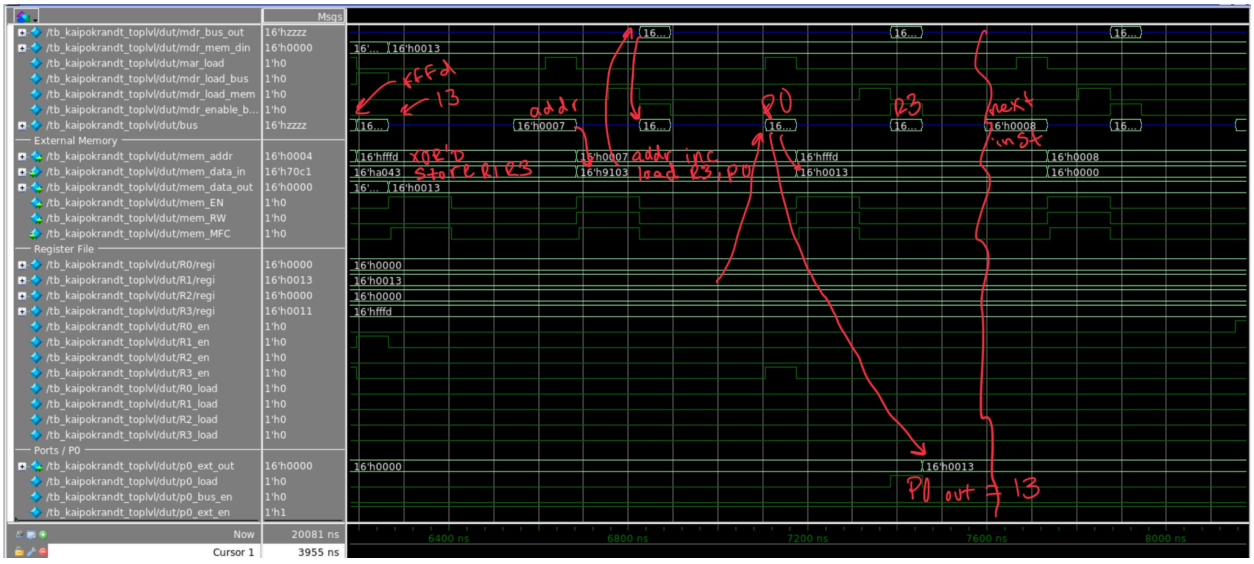


This shows the end of XOR and the beginning of STORE.



The STORE R1,(R3) instruction above shows the bus containing 0xA043, the STORE R1,(R3). Soon after this instruction, it carries 0xFFFD, which is R3's current value. This is used as the memory ADDRESS! MAR loads 0xFFFD, and then mem_addr reflects that. After this, the bust shows 0x0013 as R1 drives its value to the MDR and into mem_data_in for the write.

Kai Pokrandt
System Synthesis



This is the final instruction, LOAD(R3),P0. The bus carries 0xFFFD as the address, then later carries 0x0013 as the value read from memory. When MDR drives the bus with 0x0013 and the MEM FSM sets the destination load signal for register index related to P0, P0 captures 0x0013 and is driven to p0_ext_out.

These waveforms show the behavior of the bus throughout the whole sequence, and demonstrate that the microcontroller implements the defined instruction set and that all data moves appropriately. These screenshots above are chronologic.

Analysis & Conclusion:

This project validates the single-bus microcontroller architecture in a synthesizable way. The modular design with separation between all modules helped tremendously with debugging. Each block could be independently worked with and tested to see if issues arise. This also helped synthesis not break the whole thing without knowing what went wrong. Functionally, the demonstration testbench shows all required features, including immediate and register-register ALU ops, register moves, memory addressing, external port writing, etc. The final result P0 matched mathematically to what the output should be, proving that all pieces of the data path and control path work together as intended pre and post-synthesis. This project met expectations. The waveform analysis proves this. I had tremendous issues with wiring up my toplevel correctly, and had countless bugs post synthesis I had to fix. There were a lot of race conditions that I had to solve once complex timings were involved. The first 2 steps were fairly simple. Setting up the FSMs was challenging and took a long time. Once I got it all wired up, I had numerous issues with my pre-synthesis design that caused extreme issues with my post-synthesis waveform. I had hardcoded delays in my memory that fully broke the entire module and took me several hours to find. I also had issues with my bus running multiple sources at once, causing X state for my P0 output. I also had a bug with my MAR, where I was tri-stating the output for absolutely no reason. Interpreting and resetting my waveforms over and over post-synthesis ate up most of my time due to just setting up the ModelSim again and again.