

- Arithmetic: add, addi, addu, sub, subu
- Logical: and, or, sll
- Data Transfer: lw, sw
- Conditional Branch: beq, bne, slt, sltu

We successfully implemented this project through a structural design approach, first implementing the required lower level components (such as the various types of multiplexers), and then combining them to form the higher-level components (such as the register file and control logic). Finally, we wired up these higher-level components together in a fashion similar to the diagram in *Figure 1* to produce the highest-level single-cycle processor component named *mips_single_cycle*.

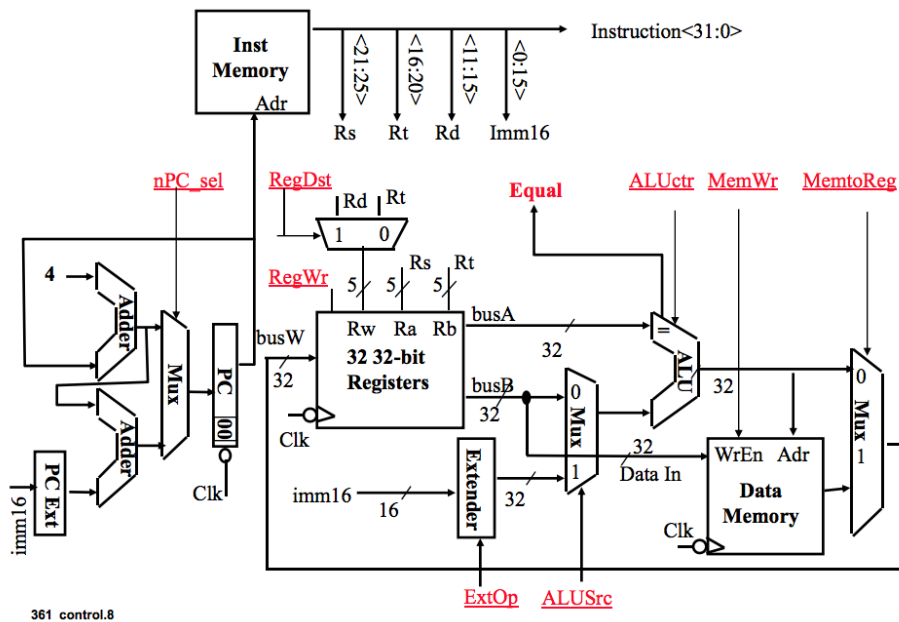


Figure 1: Single Cyclor Processor Design (Slide 8, Lecture 8)

We justify these design decisions through the benefits that this approach provides, which are mainly:

1. A modular approach allows us to not only test and debug individual components, but also allows us to reuse components and work as a team building multiple components at the same time.
2. An ALU from previous projects was reused in this project, in large part because of its applicability to many operations in the requested subset of the MIPS Instruction Set.
3. Control logic is wired manually using a control logic component and built from very basic gates as a custom logic component for this very specific set of instructions. This is to ensure speed and output according to specification on the project handout design document.

Control Signals

The results of our project include all VHDL files implemented and used in making the processor, as well as the waveforms of the test benches running the three test *.dat files. A mapping from VHD file to component can be found below:

Component	File Name
Control Logic	control_unit.vhd
PC Instruction Logic	pc_logic.vhd
ALU Control Logic	alu_control.vhd
ALU	alu.vhd
32 32-bit registers	registerfile32.vhd
Extender	extender.vhd
Data Memory	sram.vhd (library file)

Furthermore, we have outlined the control signals in a table below:

ALU Operations

Operation	Code	func	Control Signal
Load/Store	00	X	000
Branch	01	X	001
Add	10	100000	000
Subtract	10	100010	001
And	10	100100	010
Or	10	100101	011
SLT	10	101010	101
SLTU	10	101011	110
SLL	10	000000	100
AddU	10	100001	000
SubU	10	100011	000
AddI	11	X	000

Control Unit Operations

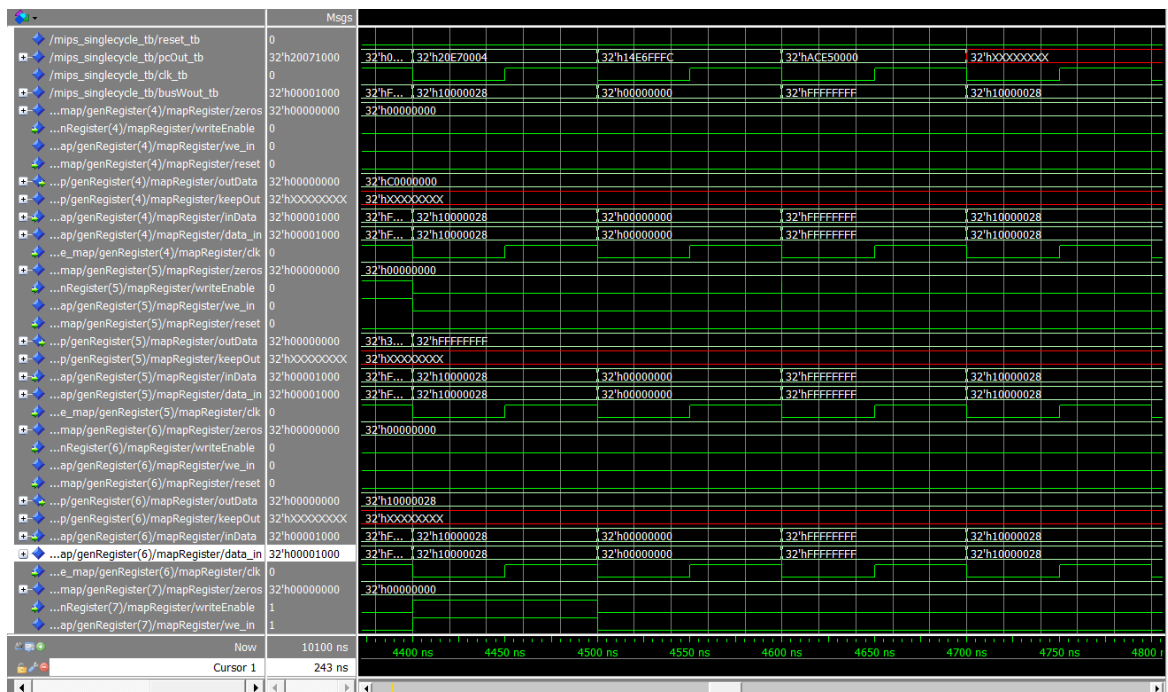
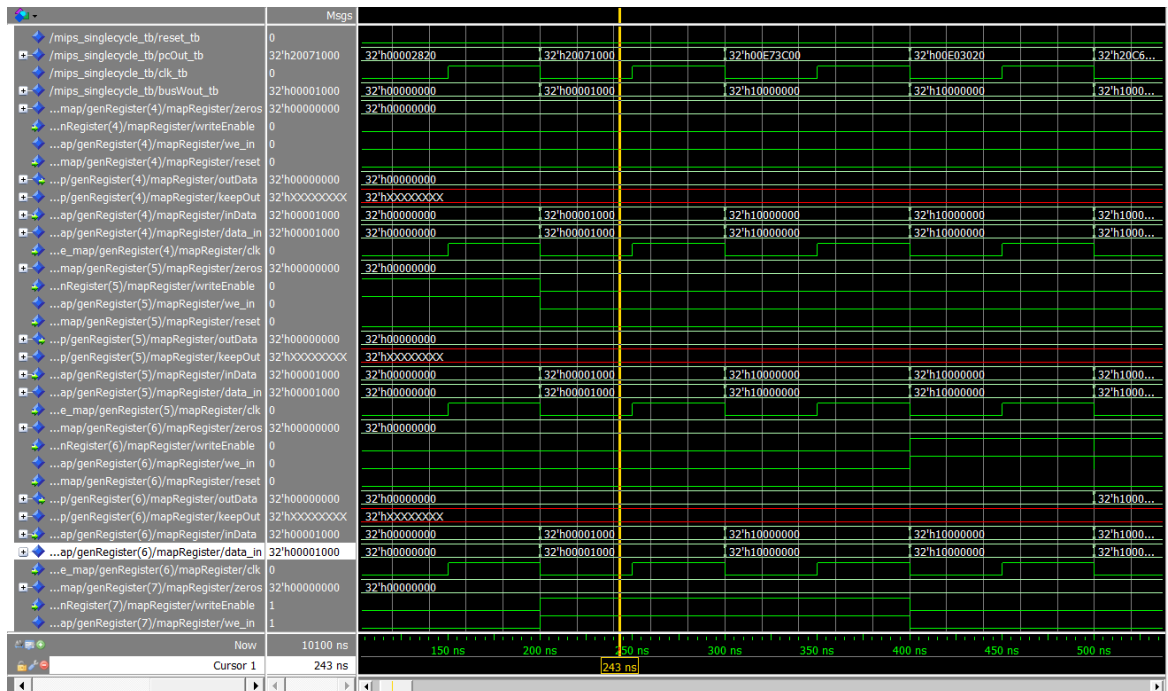
Operation	Opcode	ALU Op	RegDst	RegWr	ALU Src	PC Src	MemRead	MemWr	MemReg	ExtOp
R Type	000000	10	1	1	0	0	0	0	0	1
BEQ	000100	01	X	0	0	1	0	0	X	1
BNE	000101	01	X	0	0	1	0	0	X	1
AddI	001000	11	0	1	1	0	0	0	0	1
lw	100011	00	0	1	1	0	1	0	1	1
sw	101011	00	X	0	1	0	0	1	X	1

As indicated above, the various operations that our processor must perform will require the use of one or more control signals listed above. For example, an R-Type instruction will have an opcode of 000000, and as a result will flag various controls on and off to components located around the processor. Note that for all components, the *ExtOp* signal will always be set.

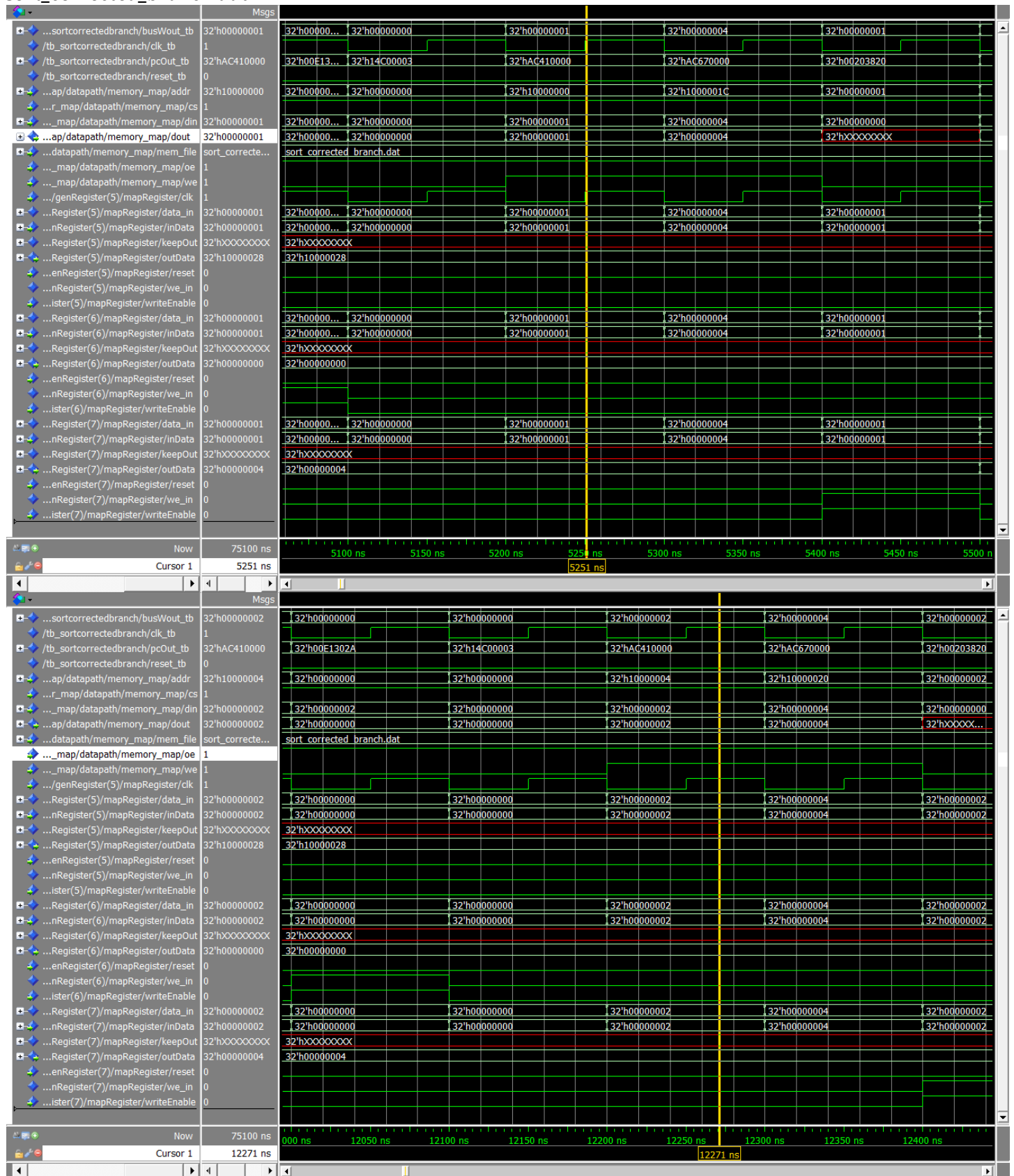
Waveforms

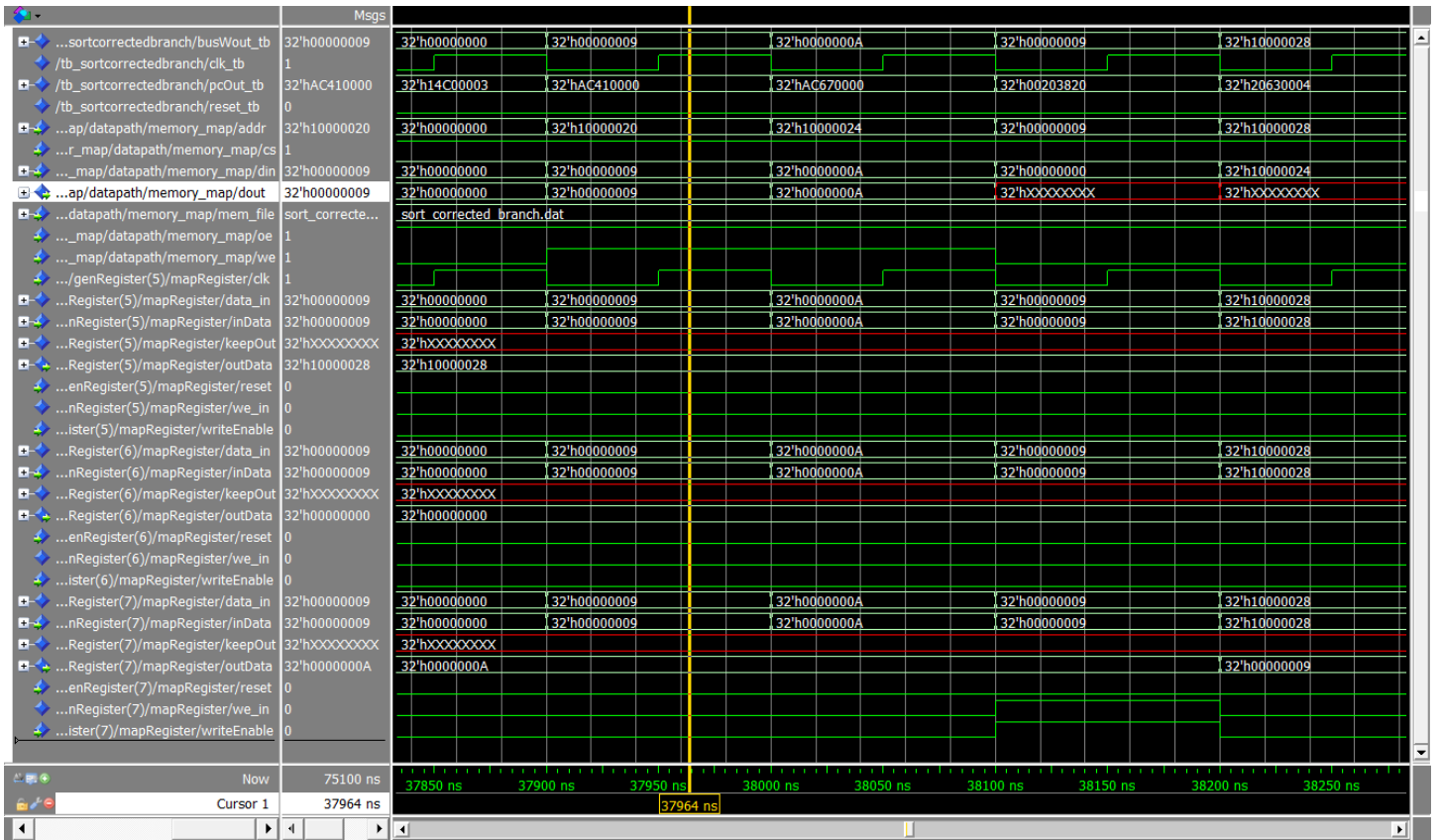
We attach the significant waveforms for the three test *.dat files in this section. These are the waveforms that show results, either by writing to memory the correct 32 bit hex value at the correct address, or by outputting that correct result to the output port of the processor. Both types of correct output can be seen in the waveforms below.

unsigned_sum.dat, showing first and last segments of waveform trace.

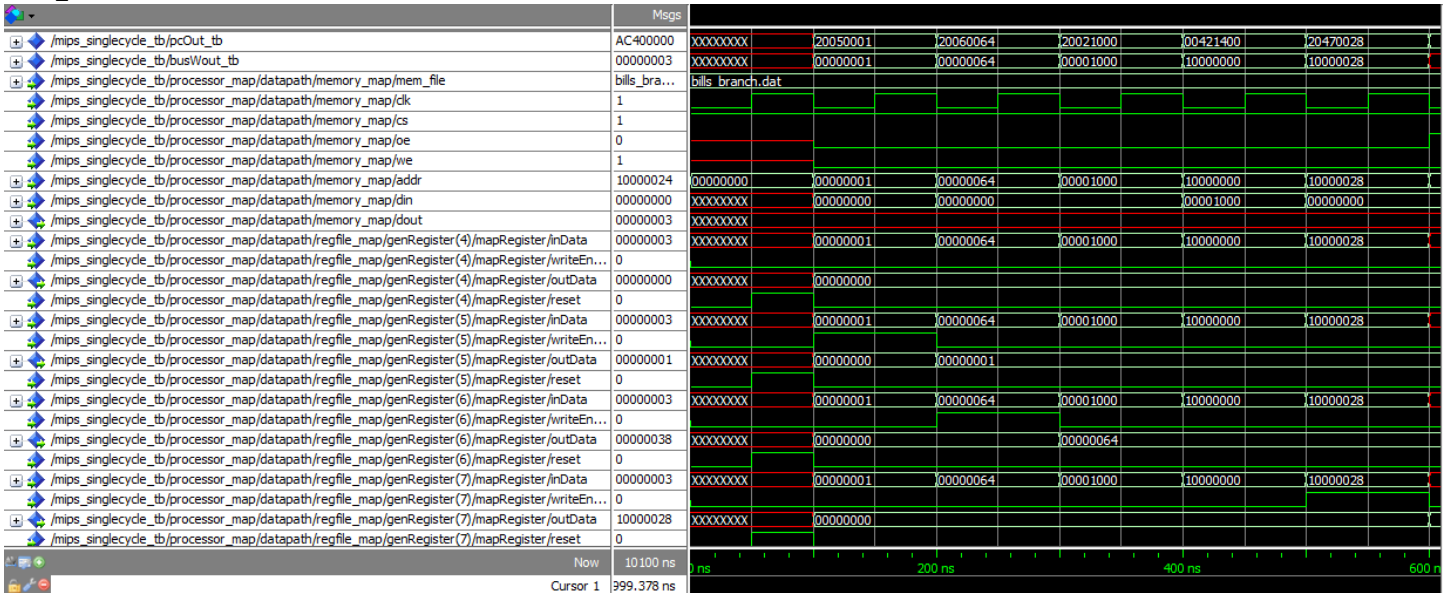


sort_corrected_branch.dat





bills_branch.dat



The figure displays three sequential screenshots of a logic analyzer interface, showing digital signals for a MIPS single-cycle processor. Each screenshot includes a list of signals on the left, a 'Msgs' column, and a waveform area with a hex data table and a time-domain plot.

Screenshot 1 (Top): Shows the initial state of the processor. The 'Msgs' column contains the address 'AC400000'. The hex data table shows the instruction 'bills_branch.dat' at address 10000004. The time-domain plot shows a signal transition at 1000 ns.

Screenshot 2 (Middle): Shows the processor executing a branch instruction. The 'Msgs' column contains the address '1447FFF9'. The hex data table shows the instruction 'bills_branch.dat' at address 10000009. The time-domain plot shows a signal transition at 1400 ns.

Screenshot 3 (Bottom): Shows the processor completing the instruction. The 'Msgs' column contains the address 'AC400000'. The hex data table shows the instruction 'bills_branch.dat' at address 10000004. The time-domain plot shows a signal transition at 2000 ns.

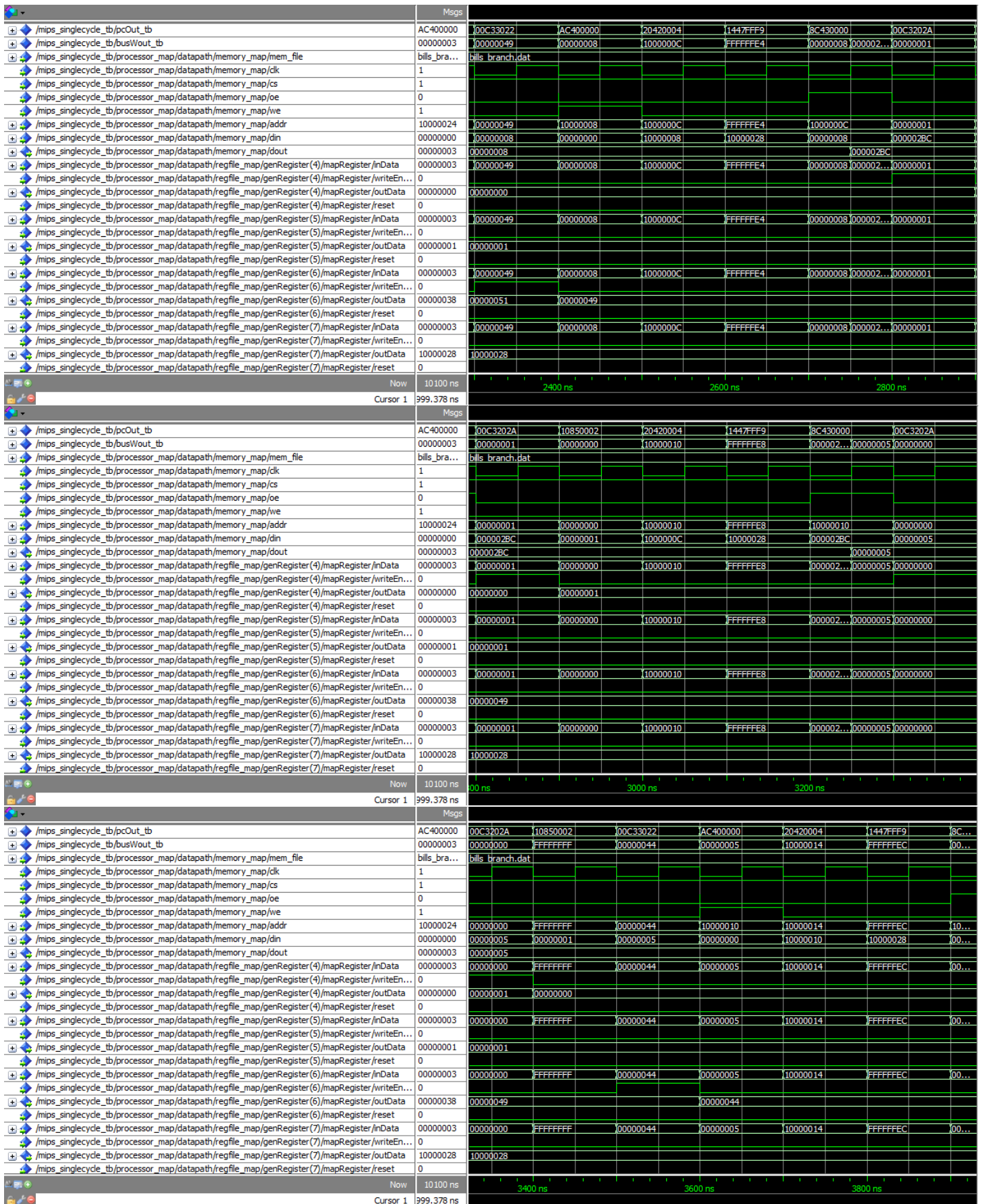
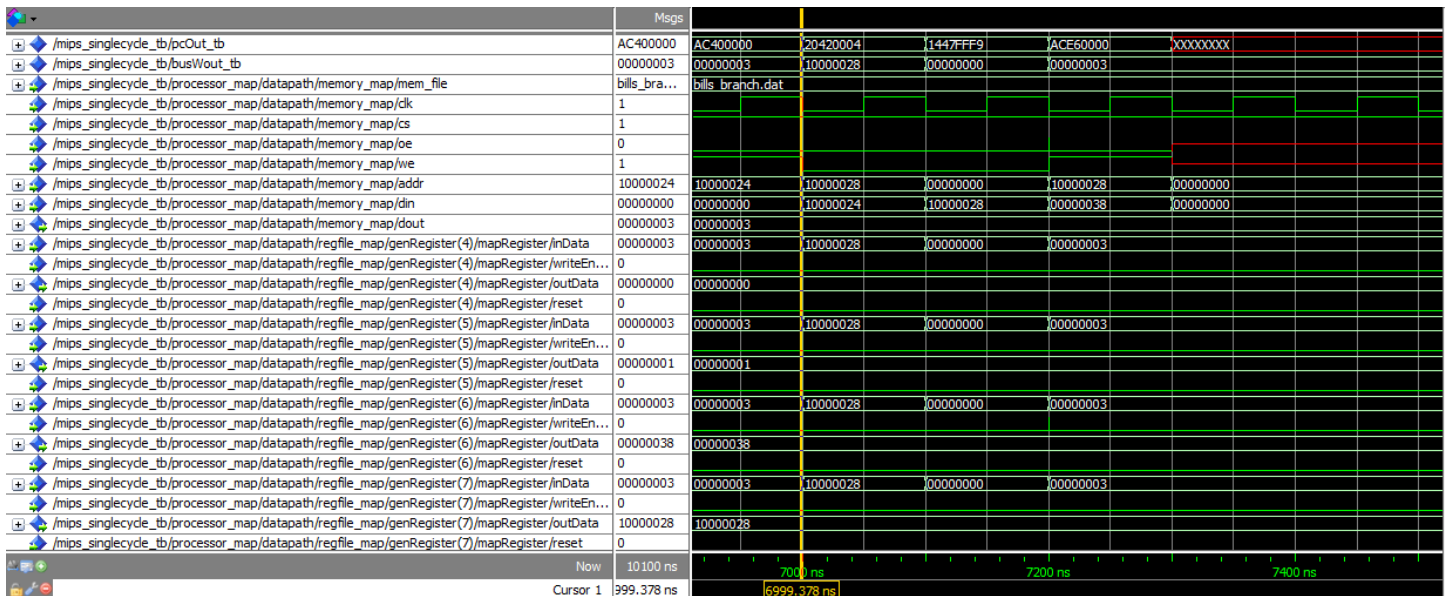


Figure 10: Waveform of the processor and memory components. The figure displays three sequential timing diagrams, each showing the activity of various processor and memory components over time. The components listed on the left include:

- `/mips_singlecycle_tb/pcOut_tb`
- `/mips_singlecycle_tb/busWout_tb`
- `/mips_singlecycle_tb/processor_map/datapath/memory_map/mem_file`
- `bills_bra...`
- `/mips_singlecycle_tb/processor_map/datapath/memory_map/clk`
- `/mips_singlecycle_tb/processor_map/datapath/memory_map/cs`
- `/mips_singlecycle_tb/processor_map/datapath/memory_map/oe`
- `/mips_singlecycle_tb/processor_map/datapath/memory_map/we`
- `/mips_singlecycle_tb/processor_map/datapath/memory_map/addr`
- `/mips_singlecycle_tb/processor_map/datapath/memory_map/din`
- `/mips_singlecycle_tb/processor_map/datapath/memory_map/dout`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(4)/mapRegister/inData`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(4)/mapRegister/writeEn...`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(4)/mapRegister/outData`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(4)/mapRegister/reset`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(5)/mapRegister/inData`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(5)/mapRegister/writeEn...`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(5)/mapRegister/outData`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(5)/mapRegister/reset`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(6)/mapRegister/inData`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(6)/mapRegister/writeEn...`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(6)/mapRegister/outData`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(6)/mapRegister/reset`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(7)/mapRegister/inData`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(7)/mapRegister/writeEn...`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(7)/mapRegister/outData`
- `/mips_singlecycle_tb/processor_map/datapath/regfile_map/genRegister(7)/mapRegister/reset`

The waveforms show the following time intervals and values:

- Diagram 1:** Time intervals of 10100 ns and 399.378 ns. Values include AC400000, 00000001, 10000020, FFFFFFFF, 00000000, 00000001, 00000002, 00000000, FFFFFFFF, 0...
- Diagram 2:** Time intervals of 10100 ns and 399.378 ns. Values include AC400000, 00000003, 10000024, FFFFFFFF, 00000000, 00000001, 00000002, 00000000, FFFFFFFF, 0...
- Diagram 3:** Time intervals of 10100 ns and 399.378 ns. Values include AC400000, 00000003, 10000024, FFFFFFFF, 00000000, 00000001, 00000002, 00000000, FFFFFFFF, 0...



Conclusion

To conclude, we have successfully created a single-cycle processor that performed all of the operations requested in the specification. As is evident in the above waveforms as well as by running the test bench, our processor is capable of reading a MIPS instruction set and producing valid output.

Challenges

1. Much early discussion revolved around how we should efficiently split up the work amongst our team. We found that the best collaborative method was using a private GitHub repository and assigning standalone components to each team member (e.g. one member took the registers, and the other took the control logic). After establishing these collaborative guidelines, progress was made at a much quicker pace.
2. The control logic had to be designed from the bottom up using basic gates, and it was a challenge to create a combinational structure that supported very many signals. We surmounted this by creating a truth table, forming Boolean expressions, and implementing them in logic gates (as was taught in *EECS 303*).
3. Reading from files and storing registers was something none of our teammates had encountered before, and was thus a challenge in implementing in our processor. We solved this by viewing the existing *sram.vhd* component, as well as looking at its test bench, *sram_demo.vhd*. From there, we were able to adapt the component to suit our needs.