**CHRISTIAN YENKO**

**BRUNO PEYNETTI**

**JACOB KOBZA**

**EECS 361: SINGLE CYCLE PROCESSOR, GROUP 2**

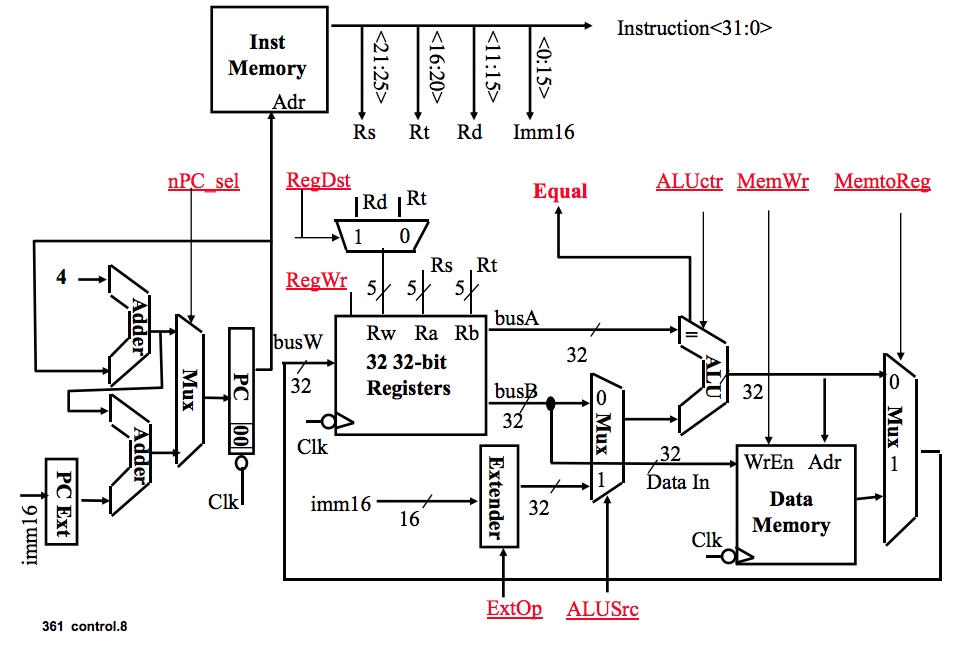
**Introduction**

In this project, we were asked to design, implement, and test a single-cycle processor that is capable of handling the below subset of the MIPS instruction set:

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

**Design**

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 Cycler 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*

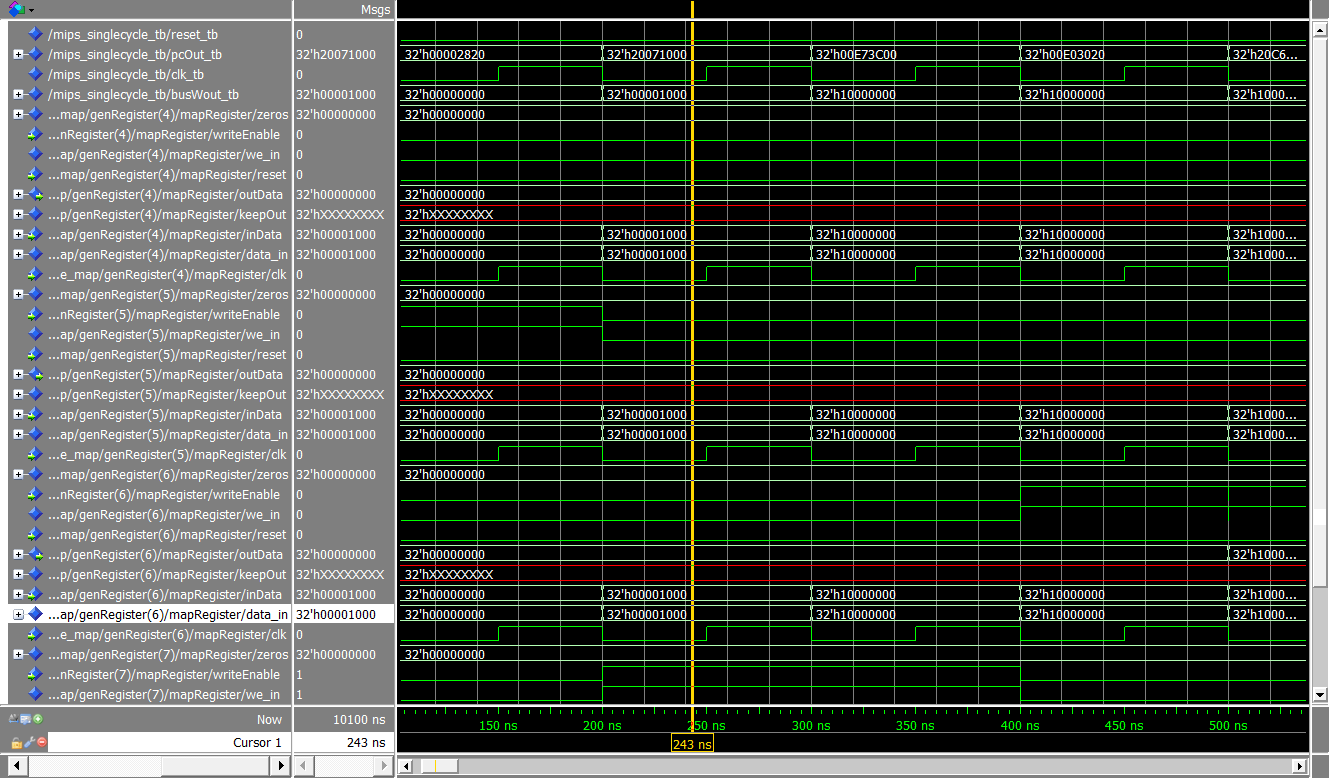
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Operation** | **Opcode** | **ALU Op** | **RegDst** | **RegWr** | **ALU Src** | **PC Src** | **MemRead** | **MemWr** | **MemReg** | **ExtOp** |
| R Type | 000000 | 10 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 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 |

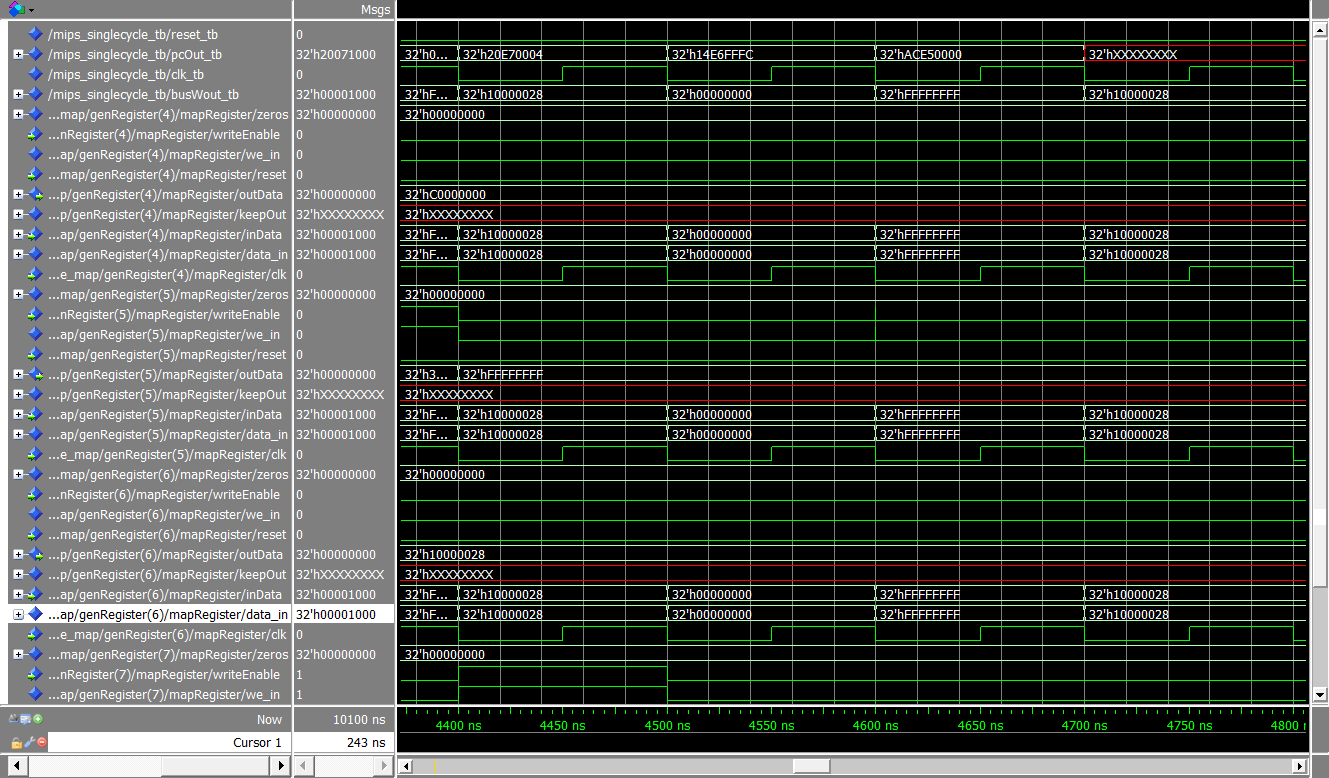
As indicated above, the various operations that our processor must perform will require the use of one or more control signals listed above. For example, an R-Type instruction will have an opcode of 000000, and as a result will flag various controls on and off to components located around the processor.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

**Waveforms**

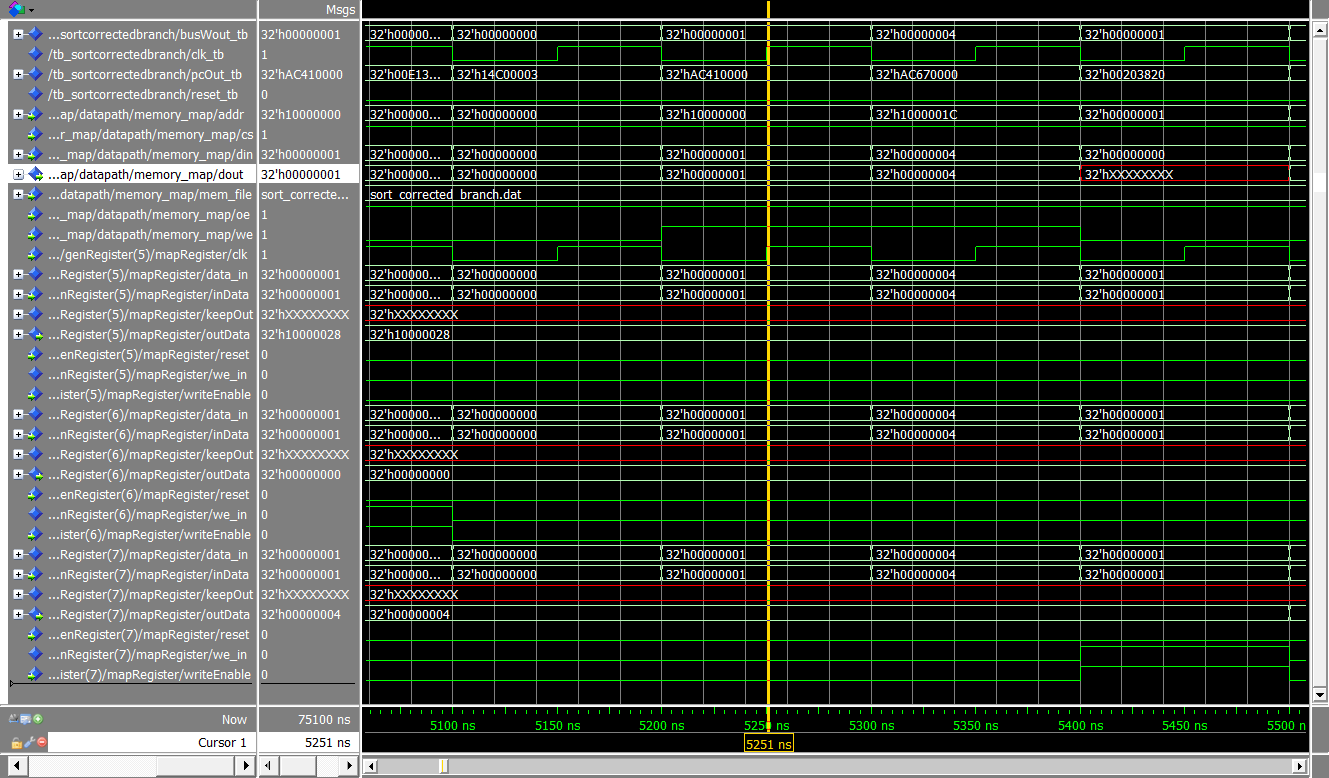
We attach the significant waveforms for the three test \*.dat files in this section. These are the waveforms that show results, either by writing to memory the correct 32 bit hex value at the correct address, or by outputting that correct result to the output port of the processor. Both types of correct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

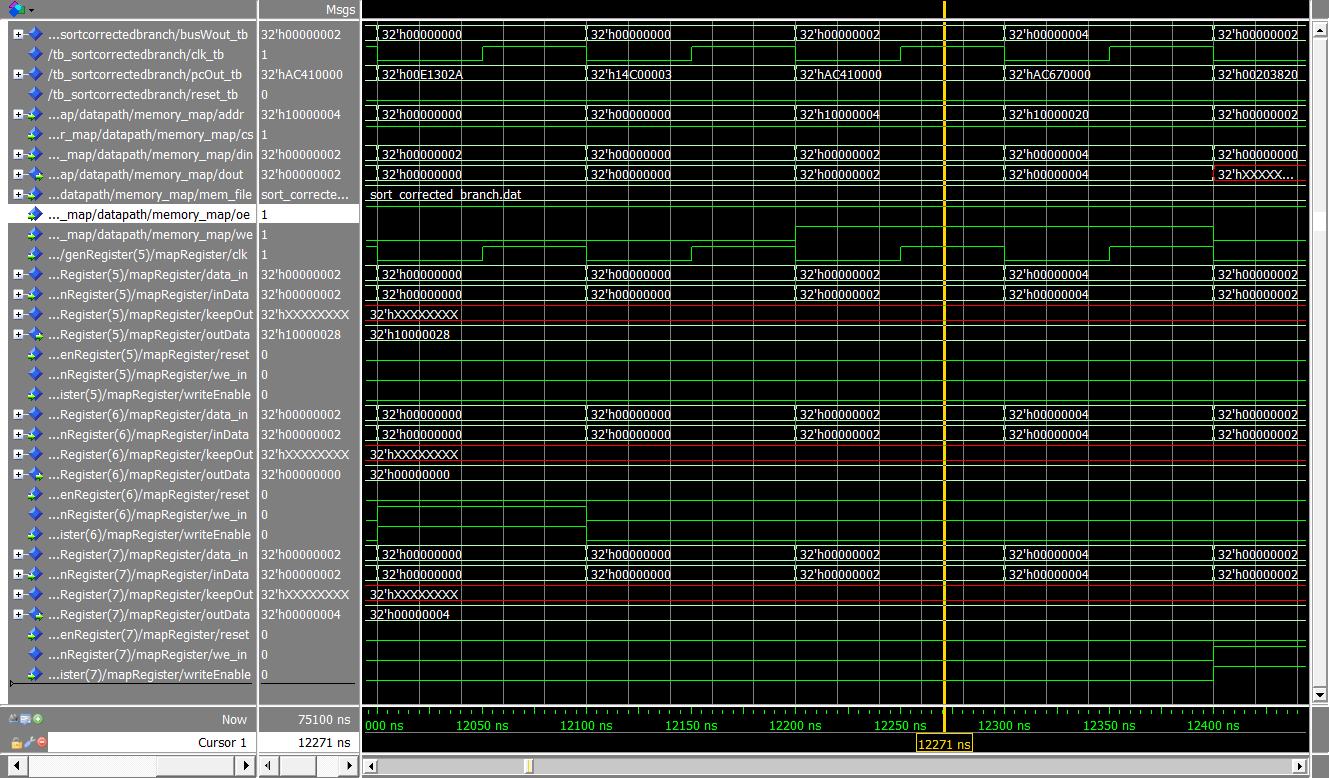
*unsigned\_sum.dat*

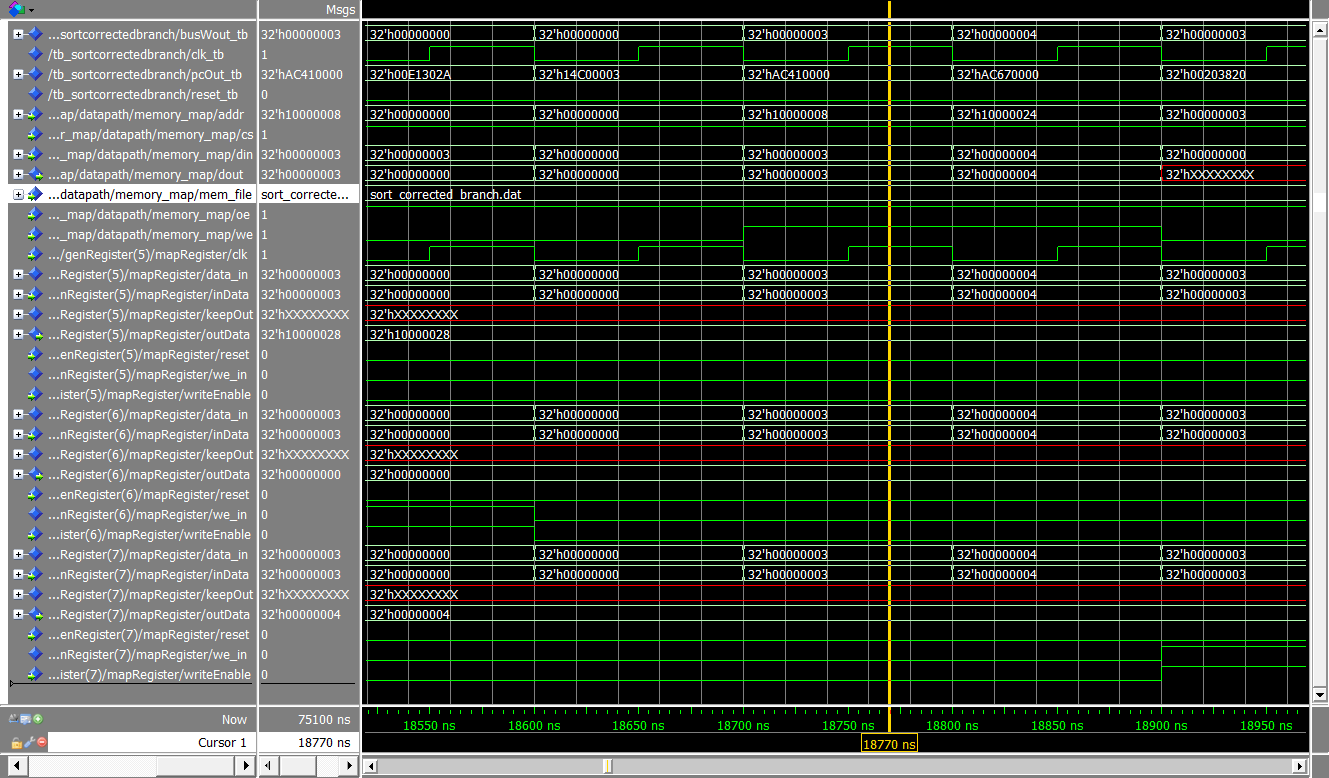


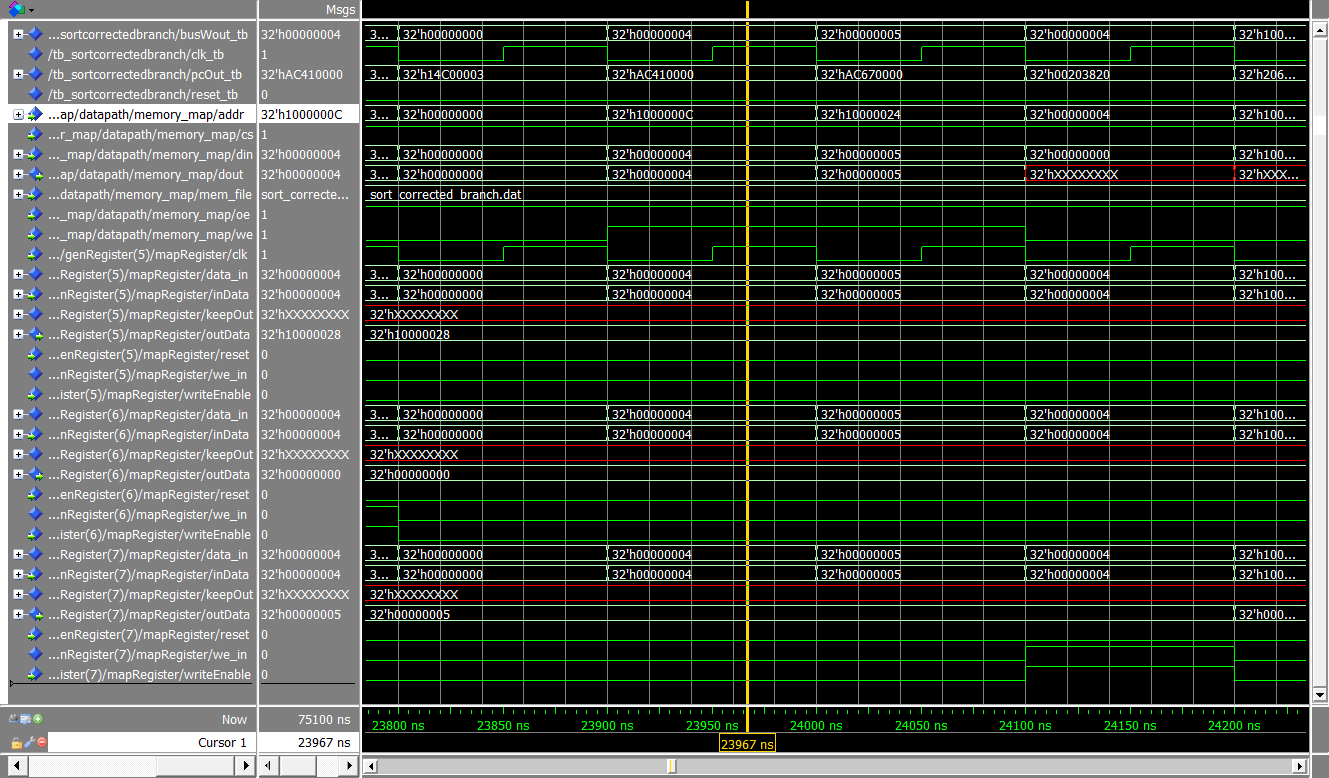


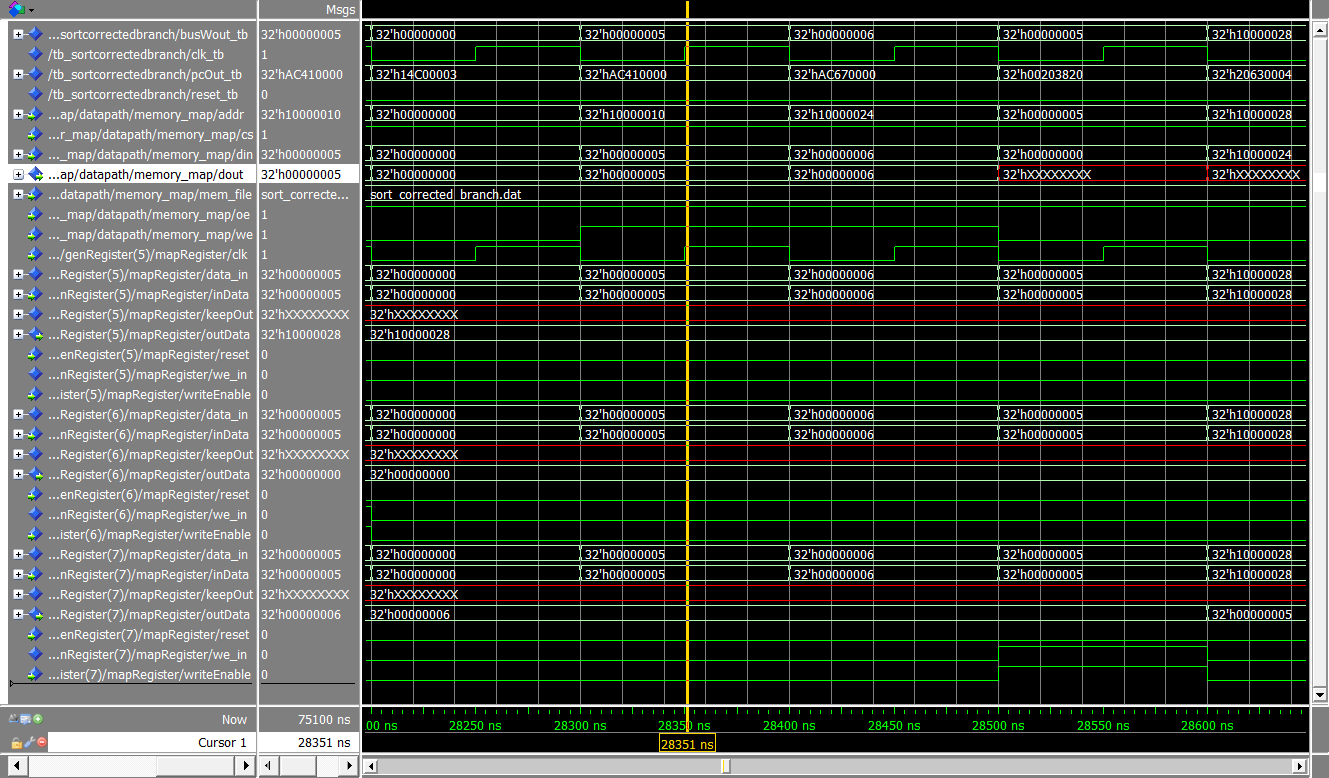
*sort\_corrected\_branch.dat*

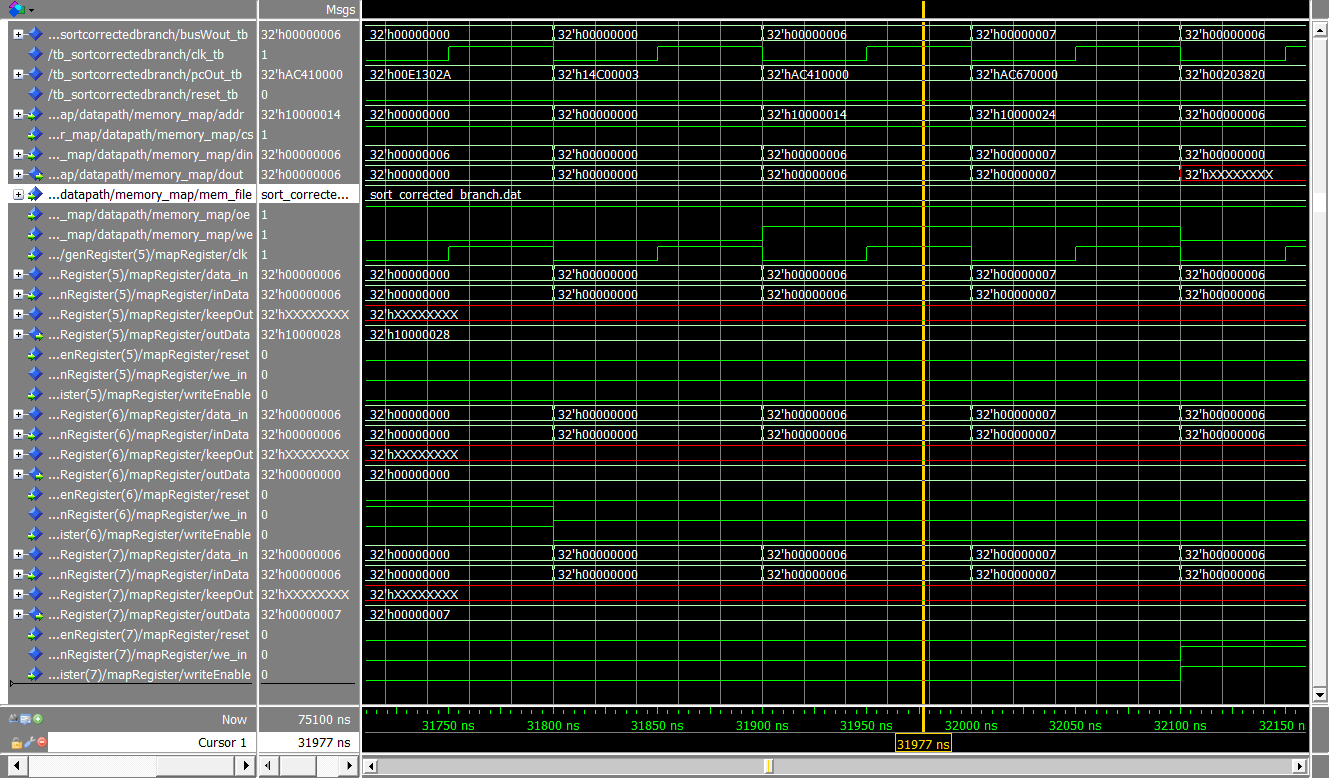


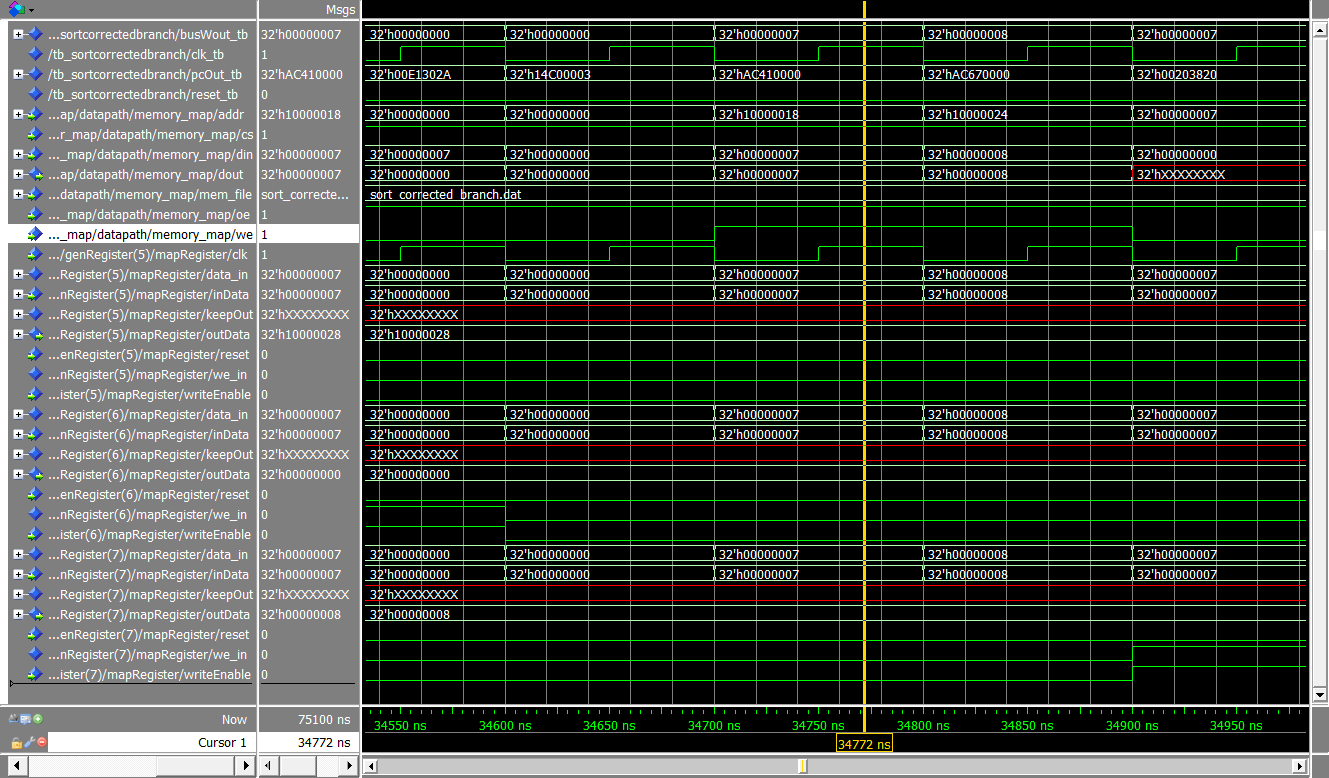


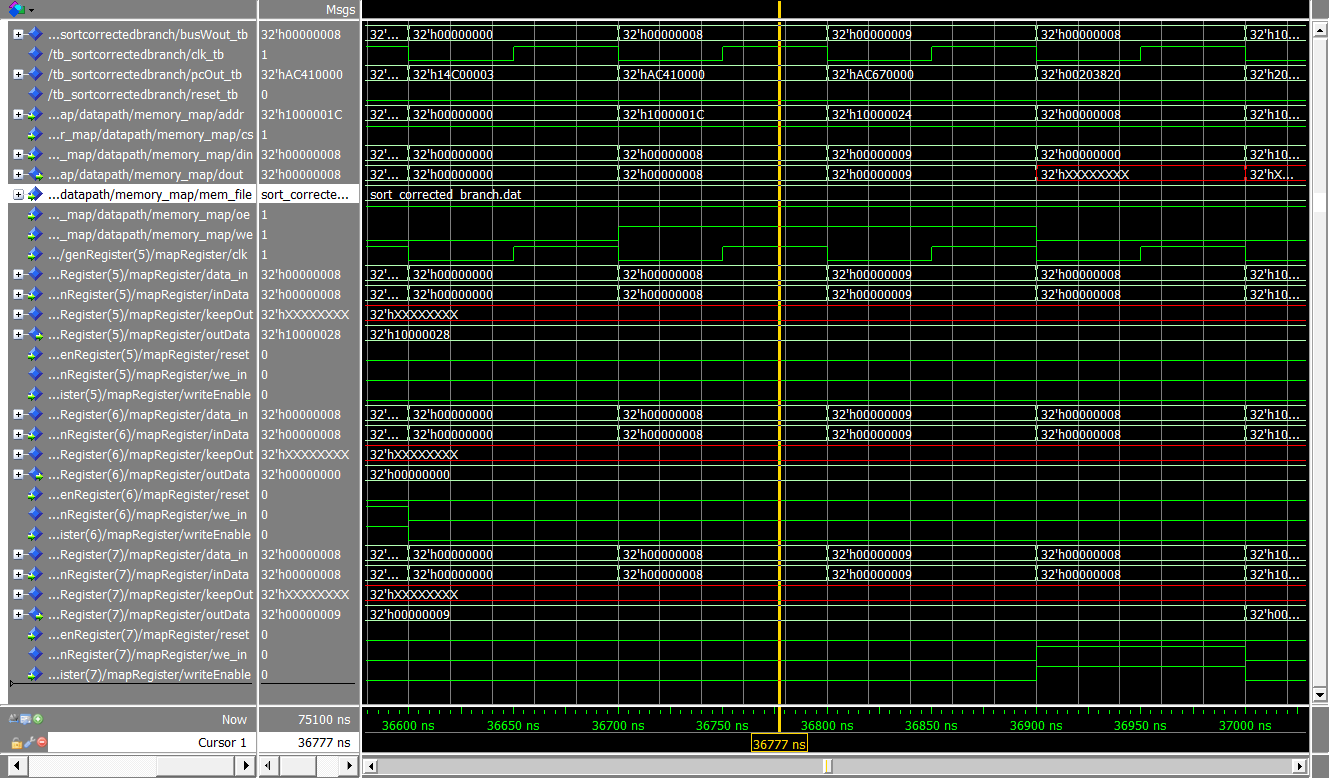


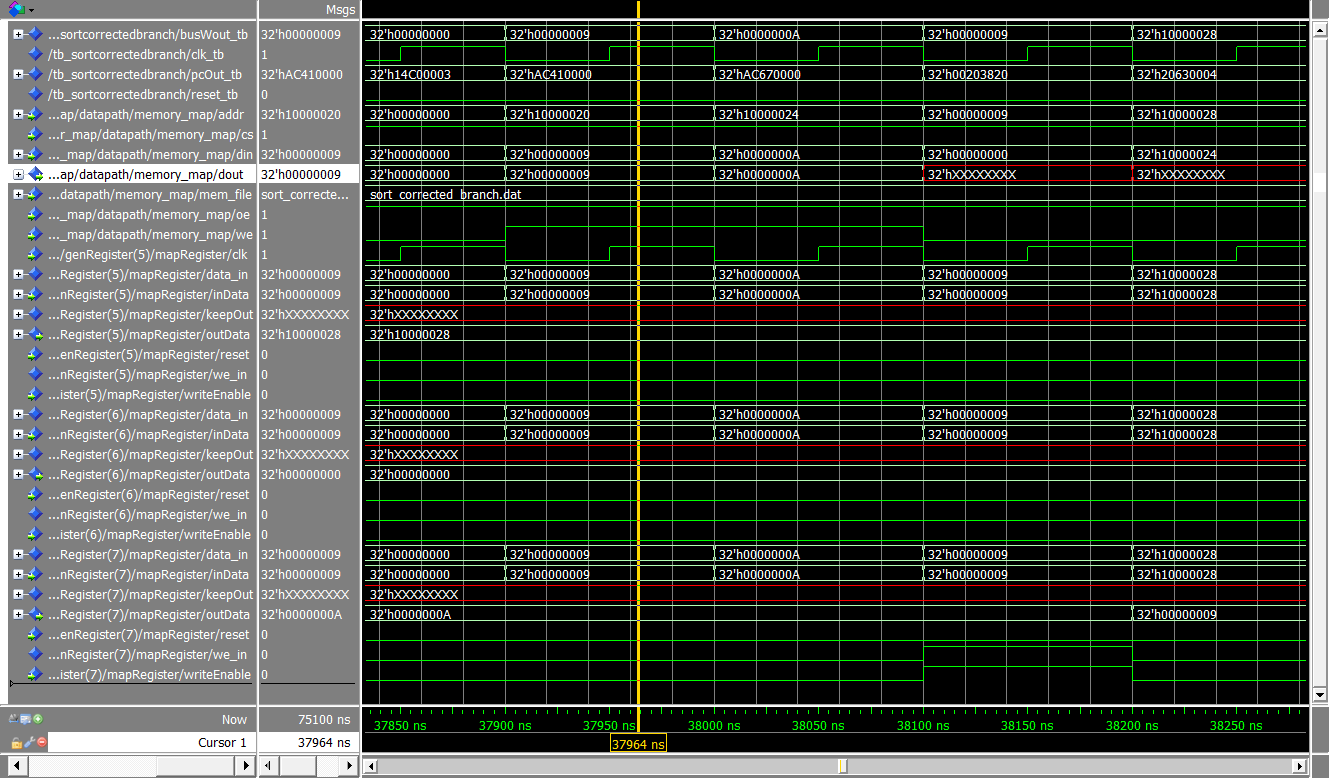












*bills\_branch.dat –* ***FILL THIS IN LATER***

**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.