

1 Introduction

For this lab, you are going to complete the implementation of your pipelined CPU. With the VHDL that is turned in with this assignment you will be able to fully simulate a subset of the MIPS ISA a second time. The CPU will run with five clock cycles for each instruction and should use all the components (not your previous CPU) that you've developed through the semester.

2 Requirements

The following requirements must be met by your CPU:

1. The assignment will be written in VHDL. The exact signature of the CPU should conform to the following definition:

```
ENTITY CPU IS
END CPU;
```

In other words, there will be no inputs or outputs for your CPU. Since you've already used this entity for your single cycle CPU, you should make a second architecture to provide an alternative implementation. This would theoretically allow someone to choose which implementation they would want to put onto a physical chip.

2. The CPU will support delay slots since they make construction of the architecture easier. It will *NOT*, however, support forwarding since that makes construction of the architecture more difficult. You may implement forwarding for extra credit.
3. The CPU must support the following MIPS instructions:
 - **add** is an R-type instruction that places $\mathcal{R}[R_s] + \mathcal{R}[R_t]$ into $\mathcal{R}[R_d]$.
 - **addi** is an I-type instruction that places $\mathcal{R}[R_s] + \text{SignExtImm}$ into $\text{RegFile}[R_t]$.
 - **beq** is an I-type instruction that places $\text{PC} + 4 + (\text{S} \ll 2)$ into PC if $\mathcal{R}[R_s]$ equals $\mathcal{R}[R_t]$. This instruction has a delay slot such that the following instruction is always executed regardless of whether the branch is taken.
 - **halt** is a new J-type instruction that is encoded as all 1 bits in the opcode. This instruction stops the execution of the processor. No other instruction should execute after this instruction. There is *NOT* a delay slot for this instruction since we can stop the processor as soon as we decode the halt. You may, however, start loading the instruction following the halt while you are decoding the halt.
 - **j** is a J-type instruction that replaces the least significant 28 bits of the PC with $(\text{A} \ll 2)$. This instruction has a delay slot such that the

following instruction is executed prior to the instruction the jump instruction refers to.

- `jal` is a J-type instruction that does the same as `j`, plus it stores $PC + 8$ into $R[31]$. We store $PC + 8$ because this instruction also has a delay slot like the jump instruction. We want to return to the next instruction that does not execute.
- `jr` is an R-type instruction that stores $\mathcal{R}[R_s]$ in the PC. This instruction has a delay slot like the jump instruction.
- `lw` is an I-type instruction that loads $\mathcal{M}[\mathcal{R}[R_s] + \mathcal{S}]$ into $\mathcal{R}[R_t]$. This instruction has a delay slot such that the value stored in $\mathcal{R}[R_t]$ is not visible to the immediately succeeding instruction. The value becomes available some time later.
- `nor` is an R-type instruction that places $\overline{\mathcal{R}[R_s] + \mathcal{R}[R_t]}$ into $\mathcal{R}[R_d]$.
- `sll` is an R-type instruction that places $\mathcal{R}[R_s] \ll \mathcal{F}$ into $\mathcal{R}[R_d]$.
- `srl` is an R-type instruction that places $\mathcal{R}[R_s] \gg \mathcal{F}$ into $\mathcal{R}[R_d]$.
- `slt` is an R-type instruction that places $(\mathcal{R}[R_s] < \mathcal{R}[R_t]) : 1 ? 0$ into $\mathcal{R}[R_d]$.
- `sub` is an R-type instruction that places $\mathcal{R}[R_s] - \mathcal{R}[R_t]$ into $\mathcal{R}[R_d]$.
- `sw` is an I-type instruction that stores $\mathcal{R}[R_d]$ into $\mathcal{M}[\mathcal{R}[R_s] + \mathcal{S}]$.

where \mathcal{R} represents the register file, \mathcal{S} represents the immediate portion of an I-type instruction that is sign extended, \mathcal{A} represents the address portion of a J-type instruction, \mathcal{F} represents the shift amount of an R-type instruction, and \mathcal{M} represents the memory.

4. All gates that are used must have four or fewer inputs.
5. Each gate should have a 35ps delay.
6. If you have a behaviorally defined control unit, it should exhibit a 90ps delay.
7. The instruction and data memories should be separate entities. This means that any modifications to the data memory will *NOT* be reflected in the instruction memory. This will prevent the architecture from supporting self-modifying code.

You should continue to use the freely available memory located at <http://www.css.taylor.edu/~jgeisler/cos381/sram.vhdl>. It provides a 64KB memory that is accessible a word at a time. You can look at the simple testbench located at <http://www.css.taylor.edu/~jgeisler/cos381/sram-tb.vhdl> for examples on how to read data from and write data to the memory unit. See the previous assignment for a review how the memory works.

3 Implementation ideas

You should implement the CPU as we designed in class. This will make it easiest to use the same components throughout the pipelined CPU as you used in your single-cycle CPU. In fact, you should probably use the sign extender and shifters that are duplicated in both architectures.

4 Deliverables

You should turn in an electronic copy of your VHDL, including all the components you created to construct the control units hierarchically. Additionally, you must demonstrate your lab to me at a scheduled time.

The project is worth 100 points as follows:

- 20 points overall structure

 - 10 points good hierarchical design

 - 10 points correct connections between components

- 70 points for CPU performing instructions correctly

 - 5 points for each instruction

 - 10 points for passing the control values through the various pipeline stages correctly