RAIK 284H: Computer Systems

# Clock

We chose to split the clock into 5 sub clocks. This allowed us to very explicitly control the order in which components were triggered. Our sub clocks are as follows:

* Clock0: program counter
* Clock1: ROM read
* Clock2: register file read
* Clock3: RAM read/write
* Clock4: register file write

# Jump/Branch logic

For our jumping and branching logic, we decided to use 3 muxes to determine what the next PC value should be. The first mux handles beq and bne instructions, the second mux handles explicit jump instructions, and the third mux handles the jr (jump register) instruction. We also decided to add a “BNE” flag to our controller to help in controlling the logic for the first mux.

# Overflow Register 7

We made the design choice to use register 7 as the overflow register from the ALU. We added a specific input into our register file from the overflow flag on the ALU and store its value when the register write clock is triggered. This allows us to access the overflow result of an instruction in the next instruction and operate on it (i.e. store to memory).

# RAM Helper

In order to allow for the user I/O board hardware components to function like memory addresses, we created a RAM Helper symbol that acts in parallel with the RAM. It takes the same input as the RAM, is clocked with the RAM, and additionally has the input pins directed into it. If reading a “memory address” from the dipswitch/buttons, the symbol will convert the signal to a vector[7..0] and output it. If writing to LEDs, the symbol will output either a 4-bit vector for one digit or a signal for the decimal point. The RAM Helper works in conjunction with a mux and the actual RAM to determine if the RAM Helper’s output or the RAM’s output should be passed along (to the next mux and potentially to the reg file).

# 7 Segment Display

We decided to write our own simple components for 7 segment display logic, instead of using a ROM and the provided mif file.We did this because we felt it allowed us to more explicitly control how we wanted to handle the interactions with the RAM (helper) and the board itself.

# Instruction Set

For this assignment, we chose to use a slightly modified version of the provided instruction set. Op Codes were chosen for commands in order to minimize the amount of logic necessary inside the processor. Similar design choices were made for the ALU codes.

|  |  |  |
| --- | --- | --- |
| Instruction | OpCode | ALU Code |
| and | 0000 | 000 |
| or | 0000 | 001 |
| xor | 0000 | 101 |
| sll | 0000 | 011 |
| Ssrl | 0000 | 111 |
| add | 0000 | 010 |
| sub | 0000 | 100 |
| slt | 0000 | 100 |
| Addiu | 0001 |  |
| Subiu | 0010 |  |
| Addi | 0011 |  |
| Subi | 0100 |  |
| Beq | 1000 |  |
| Bne | 1001 |  |
| Lw | 1010 |  |
| Sw | 1011 |  |
| j | 0101 |  |
| Jr | 0110 |  |
| noop | 0000 |  |

These OpCodes and ALU codes match up directly with the logic in our Controller and ALU components in the processor.

The instruction size is 16 bits, and the maximum number of operations allowed in a program is 255 (the processor supports 256 operations, but our assembler prepends every program with a noop in order to resolve any timing issues).

# Assembler

Our assembler was designed to read in assembly source (.s) files, and convert them to the .MIF memory file that can be set to the ROM on the Altera board. It offers full support for our instruction set, and warns of any syntax errors it detects while it parses the assembly. When the executable is run, it prompts for the file name of the assembly source (just the file name or the file name with the exception) and attempts to read this file from the current directory. It will parse the file and attempt to assemble it, giving the user feedback regarding the validity of the program by explaining errors on each line. If the assembler is successful, it will output the MIF file to the current directory.