8-Bit Computer Based on 32-Bits MIPS Processor

Krishna Thiyagarajan, Chris Brancato, and Ihsan Gunay

***Abstract*—A computer processor based on a 32-bit MIPS processor was designed. An 8-bit ISA analgous to the 32-bit MIPS ISA was made and the ISA was then implemented in Verilog with a descriptive level of abstraction. The final implementation included modules such as the arithmetic logic unit, a control unit, a data memory module, an instruction memory module, a program counter module, a register file module and the main CPU module. This report briefly describes the design of each.**

***Index Terms*—Computer Architecture, 8-Bit, MIPS, Processor**

I. Design

There are three different types of operations in our basis MIPS 32-bit architecture. The basis uses three types of instructions: R, I and J. The MIPS ISA has 4 bit addressable registers. However, the 8-bit architecture cannot handle this. Therefore, there will be 4 addressable registers, each having 2-bit addresses. One of these registers will be used as a default "results" register. Another one of these registers will be used as a stack pointer in order to manipulate the stack for storing temporary variables when necessary. This allows a maximum of four bits of operation code (opcode) instructions. Therefore, sixteen operations were selected by usefulness in a real computer. The selection resulted in following operations whose functions are described in the attached Reference Sheet: move, add, and, not, nor set less than, shift left logical, shift right logical, jump, jump and link, load word, store word, branch on equal, branch on not equal, add immediate and load immediate. The implementation of the modules in Verilog will be discussed below.

II. Arithmetic Logic Unit

The ALU was designed to do all the functions that the ISA requires. To do this, the ALU takes an input of instruction code, two inputs, in0 & in1, program counter and a clock signal. The instruction code is given to the instruction memory. The input registers addresses are implied in the instruction given and it is passed by the CPU to the ALU. The clock signal keeps the ALU in sync with the rest of the computer.

The ALU also has the ALU output, and an overflow flag. The output is the result of the operation that the ALU has performed given the instruction. The jump output is all set to the maximum unsigned number (i.e. 255) in order to indicate that a jump operation was being performed. The overflow flag is turned on when there is an overflow in the operation that the ALU performs.

Initially, the ALUs jump variable is set to 0. After that , at the positive edge of a clock cycle, the ALU will take all the inputs given to it and will perform the above 16 operations. Since each was done with a descriptive abstraction, a case statement was set up to determine the opcode and the output was directly related to the inputs as a descriptive assignment. For example, shift left logical was implemented as follows: $out = in1 << imm2$, $overflow = 0$ and $jump = 8'b0$

III. Control Unit

The control unit is what directs the operation of the processor. It turns on units and turns off units when necessary. Our design of the control unit receives the instruction and the clock as inputs and outputs memory read/write enable, register write enable, select write source and register addresses to read and write from. The control unit then activates on the positive edge of a clock cycle and determines what operation the CPU will be doing at that specific cycle. Once that is determined, the control unit determines what registers the outputs will be going to and will provide the correct controls.

For example, if the operation to be performed is addi, then the control unit recognizes that the final register is the first register that is indicated and appropriately defines register one, register two and register write.

IV. Data Memory

Data memory deals with the stack, the heap and data manipulation. The data memory module takes in data address, write data, write enable and the clock signal as inputs. It outputs the read data, the data that was read from the memory if that is necessary. The stack, heap and data are all stored in a text file. The data memory module initially reads the data into an internal 8 bit by 191 bit register file. Then, at every positive clock edge, it simply checks whether the write enable is activated and writes to the given data address. If write enable is not activated, it reads the data and outputs the read data to the defined output register.

V. Instruction Memory

Instruction memory deals with the instructions that are being fed to the CPU. Similar to the data memory module, the instruction memory module takes in instruction address and clock signal as inputs and outputs the instruction data. Initially, all the instructions are read from the instMem.bin text file which contains the compiled binary instructions for the program. Then, at each positive clock edge, the memory is read at the instruction address as given by the program counter and the instruction data is outputted.

VI. Program Counter

The program counter module keeps track of which instruction to execute from the instruction memory module. It takes in jump offset, the clock signal and the pc control from the control unit module as inputs. It then manipulates these to output a certain program count to the instruction memory as appropriate. Initially, the program count is set to zero as that is where the first instruction will be. After that, at every positive clock edge, the program count will be updated to the next binary number if the pc control fed to the program is 8 bit binary zero. If it is not 8 bit binary zero, a jump offset will also be added to the initial increment of the program count and the new program count will be outputted.

VII. Register File

The register file keeps all the registers used in the program. It takes in two register addresses, a write register address and write data and write enable as inputs. It outputs two 8-bit numbers indicating register data. Initially, the register sets every register in the register file to zero. Then, when writing to register is enabled, the register file sets the register at the write address to the inputted write data. At every cycle, the output registers are set the numbers in the register files at the two inputted register addresses.

Assembler

To make the task of converting from our ISA assembly code to binary code, an assembler was designed. The assembler takes in each line of a text file that contains the assembly code and translate them into binary code by splitting it up as strings and parsing them. The following four pseudo codes were also added to the assembler to make it easy to do some immediate operations. The addl pseudo code stands for add large and can add immediate negative and positive numbers. The lil pseudo code stands for load immediate large and can load a specific number to a certain register. The slll pseudo code stands for shift left logical large and shifts by amounts more than 3. The srll pseudo code stands for shift right logical large and shifts right by amounts more than 3.