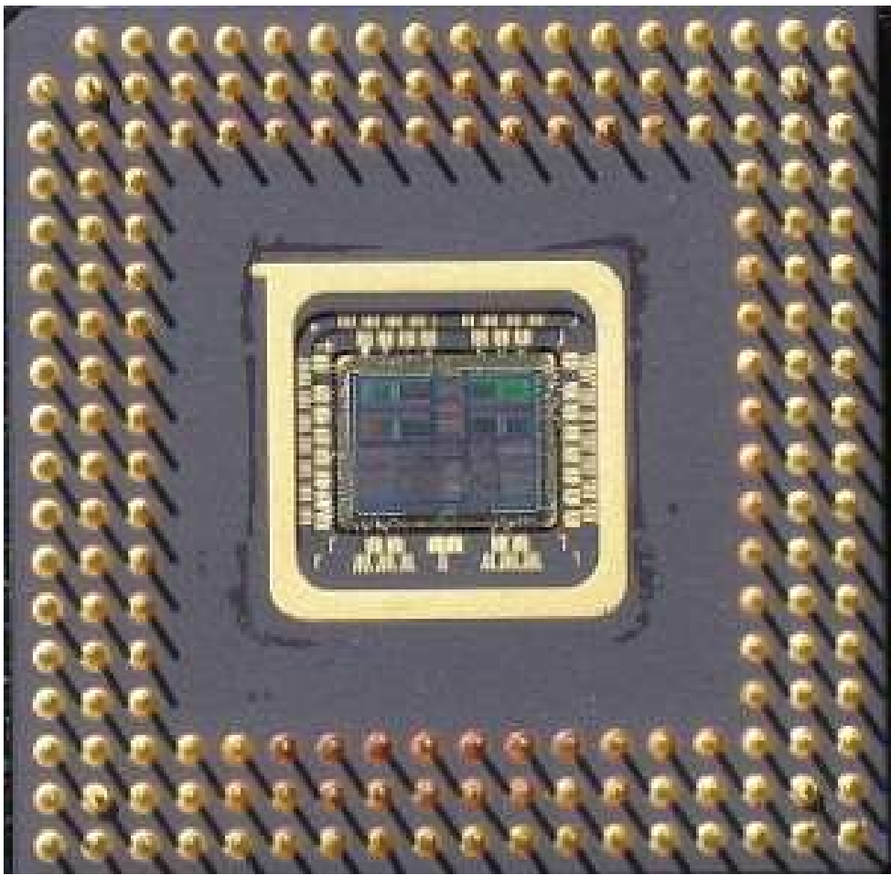

Single-cycle MIPS CPU + Extended MIPS CPU

Kevin Espinoza

First, I will explain my driver code. It is structured so that this is where the filename is received as input and the register file and memory are initialized. There is a **CPU** object here called cpu that can call **fetch()** and takes the user inputted filename as a parameter. **fetch()** is used as the loop control variable in a while loop (the loop knows when to stop because **fetch()** returns false when it has reached the end of file). After **fetch()** returns, the driver code is responsible for printing clock cycles (**getTotalClockCycles()**), what registers/memory were touched (**printTouchedRegister()** and **printTouchedMemory()**), if registers/memory were touched (**registerWasTouched()** and **memoryWasTouched()**), the pc value (**getPC()**), and total execution time (**getTotalClockCycles()**).

Next I'll give a brief overview my **CPU** class: it contains all the required member variables and some unique ones I added to give the functionality I needed. **std::vector<std::string> registerfile_name** stores the names of all the registers in strings at their respective indices (registerfile_name[0]= $zero, register_filename[1]=$at, etc). **std::vector<int> d_mem_address** works similarly for the memory addresses. There are two pairs, **touched_register<bool, int>** and **touched_memory<bool, int>** that track if registers/memory were modified and if so, what register/memory address that was. I also added a **jlink** and **regtopc** control signal for properly directing a JAL and JR instruction respectively, and a **writeToReg** variable that will hold the index of the Rd, Rt, or $ra. I have a function **convertBinaryStrToInt(std::string)** that takes the fields from an instruction and conveniently converts them to integers. In general, **fetch(),** calls decode(), which calls execute(), which in turn calls memory(), and which finally calls writeback(). When **writeback()** returns, it returns all the way back to decode() where it figures out the next PC addresses and increments **total_clock_cycles**.

When **fetch()** is called by the driver code, it starts by opening the file with the filename provided by the user and iterates through the file until it reaches line of the text file that is equal to **pc** / 4 (for the first instruction 0 / 4 = 0 would make it read the first instruction) and extracts it as a string. After fetching an instruction, it "increments" **pc** by 4 and stores that into **next_pc**, then calls **decode(),** passing the instruction as a string. There are more instructions that follow but we will return to these later, as these do not execute until **decode()** returns.

decode() has a local variable for every possible field that could be extracted from any of the ten, 32-bit MIPS instruction that my application supports(namely **opcode**, **rs**, **rt**, **rd**, **shamt**, **funct**, **imm**, and **addr**). Each substring is extracted from the instruction and converted and stored as an integer in the corresponding variable. I chose to call the **controlunit()** here to set the control signals since this happens to be the first place it becomes relevant. The conditional statement that follows (we are skipping resetting **regtopc** = 0 for this first part) represents the mux that determines the write register number in the register file. If regdest is set (equal to one), **rd** becomes the register for the write register number, otherwise it's **rt**. **imm** is then **signExtended()**, and **jump_target** is calculated right after by performing a **shiftLeftTwo()** on the address and adding the appending the four most significant bits from the PC after converting those to integers with **fourMostSigFromPC()**. **execute()** would be launched right after this instruction, but only if there is no **jump** control signal. This is expected behavior because a jump instruction does not need to access **execute()** to perform an ALU operation, **memory()** to access a

memory address, or **writeback()** to writeback to a register. After this is a call to **execute()**, passing only the variables it will directly need.

Both the **rs** and **rt** registers are passed to **execute()**, as well the **signextend**ed value, and the **funct** field. Here the **insttype** is calculated by combining **insttype1** and **insttype0** and it is passed into **alucontrol()** along with the **funct** field to determine the upcoming **alu_op** to be performed (the **funct** field is only necessary if the instruction passed is an R-Type). Local variable **alu_result** stores the result of the ALU operation performed on **alu_operand1** and **alu_operand2** (**alu_operand1** is always set to **rs**). However, **alu_operand2** is dependent on control signal that goes into the mux that feeds right into it. This is represented by the conditional statement that sets it to the **signextend**ed value if the **alusrc** control signal is flipped on, or to the value at the **rt** register otherwise. The actual ALU operation is determined by the output of the **alu_op** produced by the **alucontrol()** earlier. **alu_zero** is then set if the result of the operation was 0, which means that a branch should occur if this were a branch instruction (otherwise **alu_zero** is not set). And if the **branch** control signal was also flipped, then the **signextend**ed value that was passed into **execute()** would be **shiftLeftTwo()** and added to next_pc to calculate the **branch_target** (after which it would return). We will proceed as if this was not the case and continue to call **mem()**.

In **mem()**, both the **alu_result** and the **rt** register are passed. If the **memread** control signal is flipped, then that means **alu_result** holds a memory addresses that a value will be loaded from written to a register in **writeback()**. If instead the **memwrite** signal is flipped, then that means the **alu_result** is still an address, but one will be written to in this stage (and return right after). If neither signal is flipped, then the **alu_result** holds a calculation (implying this was an R-Type instruction) and calls **writeback()** with it. A quirk of my code is that I pass a dummy value 999 as the second parameter for this case. This is because **writeback()** takes two parameters but an R-Type does not flip a **memread** control signal, which would provide a second value to pass in.

We enter **writeback()** with an ALU result and the value at that “address” in d_mem (again, not the case for an R-Type). A series of conditional statements checks if a **regwrite** will occur, and if so, value to be written will come from memory (**memtoreg** control signal) or will be the direct output from the ALU (we are skipping the **jlink** control signal still). Regardless of the outcome, now the **total_clock_cycles** are incremented and the function returns.

Returning takes us all the way back to the **fetch()** call that triggered this all**,** and we proceed to finally determine what value the **pc** that will actually hold. If the **jump** control signal is flipped, the pc is set to the **jump_target** calculated in **execute()** and, since the jump function had no need to go all the way to write back, **total_clock_cycles** is incremented here. If the jump signal is not flipped, then we check for both a set **alu_zero** and a **branch** signal for **pc** = **branch_target**. If these conditions were not met, then **pc** is simply **next_pc**. This function call returns true since the CPU was indeed able to fetch an instruction, allowing the driver code to execute and output the data access patterns.

My implementation of JAL, was able to make use of much of the jump instruction data path. It needed to be different in some key ways though, namely that after calculating the jump address, it

needed to write back the **next_pc** to the $ra register (**register_file[31]**), meaning that the **regwrite** control signal would need to be flipped. However, once it got to the writeback() stage, it would need a unique signal to allow only it to write back to the $ra register. Thankfully since JAL is a J type instruction, it has a unique opcode that I leveraged to generate a new control signal – **jlink**. In my implementation, if a **jlink** signal is flipped, an additional out from **next_pc** is opened to the **writeback()**. **writeback()** then takes this value and stores it in the $ra register.

Implementing JR was trickier. JR is interesting because it needs to set the **pc** to the address stored in the $ra register. This is complicated further by the fact the JR is an R-type instruction, and one that behaves very differently from most other R-type instructions, most perform an ALU operation and store the result in a register. In a JR instruction, we don't want to write back to a register. So somehow we need to set a control path to change it's data path. However, JR is an R-type, so we cannot manipulate the control signals without disrupting the behavior of the other R-type instructions. My approach was to exploit the fact that, as an R-type instruction, JR has a **func** field. With this, I chose to extend the **alucontrol ()** to output it's own control signal **regtopc** when a JR instruction is detected. This signal would control a mux, located just before the PC, that would allow read data 1 from the register file through, instead of the output of the the mux that is currently just before the PC in the current architecture. This was sufficient to implement in code (and is in fact what I wrote in my code, the function simply returns after setting **pc** = **rs**), but conceptually I realized that, since this is an R-type instruction, there would be a **regwrite** that needed to be stopped only when **regtopc** was flipped on. I believe this could be addressed with an integrated circuit that controls the whether the register file takes the regwrite control signal or the regtopc signal.