

**Purpose**

The overall goal of Assignment 7 was to extend the design of the single-cycle MIPS processor from Assignment 5 and 6 to support six new instructions: MULTU, MFHI, MFLO, JR, JAL, SLL, and SLR. This lab report covers part 2 of the assignment; that is, the design of the extended datapath with support for the aforementioned instructions and an updated control unit design to drive the extended datapath. In a future report, we will implement the design, functionally verify the design through simulation, and validate the design on an FPGA device. The two tasks for part 1 of Assignment 7 are as follows: 1) Create a draft of the extended MIPS microarchitecture and 2) create a truth table for the control unit that supports the new extended MIPS microarchitecture.

**Design Methodology**

The initial task included a draft of the microarchitecture design which includes the extended MIPS instructions. Initially, confirmation was required that the microarchitecture design would include the basic framework from Assignment 5. The microarchitecture design with extended MIPS instructions can be found in Appendix A. The modules of the design were placed accordingly per instruction by developing the control unit (CU). The CU inputs include the opcodes for the extended MIPS instructions. The outputs of the CU correspond to the modules required to perform the functionality for each instruction. The inputs and outputs of the control unit are depicted in Appendix B. The control unit is shown in Figure H of Appendix A.

**SLL and SLR**:

SLL and SLR are instructions that shift an input data left or right, respectively, by a given shift amount. MIPS supports these operations as R-Type instructions. The machine code for an SLL or SLR instruction includes fields for a register source, a register destination, and a shift amount (shamt). With this in mind, we designed a module dedicated to shifting an input left or right depending on an input control signal, sl\_or\_sr, and a shift amount given by the shamt field in the input instruction provided from the instruction memory. If the sl\_or\_sr control signal is a 0, then the shifter performs a shift left by the given shamt. Likewise, if sl\_or\_sr is a 1, then the shifter performs a shift right by the given shamt. This control signal is provided by the control unit depending on the function field of this R-Type instruction. For example, if the function field is 00hex, then the instruction is a SLL and sl\_or\_sr will be. SLR gives a function field of 02hex indicating the control unit should output a 1 for the sl\_or\_sr signal. The 32 bit result of shift is fed into a multiplexor (mux) along with an output from a MFLO/MFHI instruction which will be explained later. The general idea is that the mux will select which output is eventually fed back to the register file depending on the instruction being executed. Figure A below shows the design of this module. Integration with the rest of the datapath is shown in Appendix A.

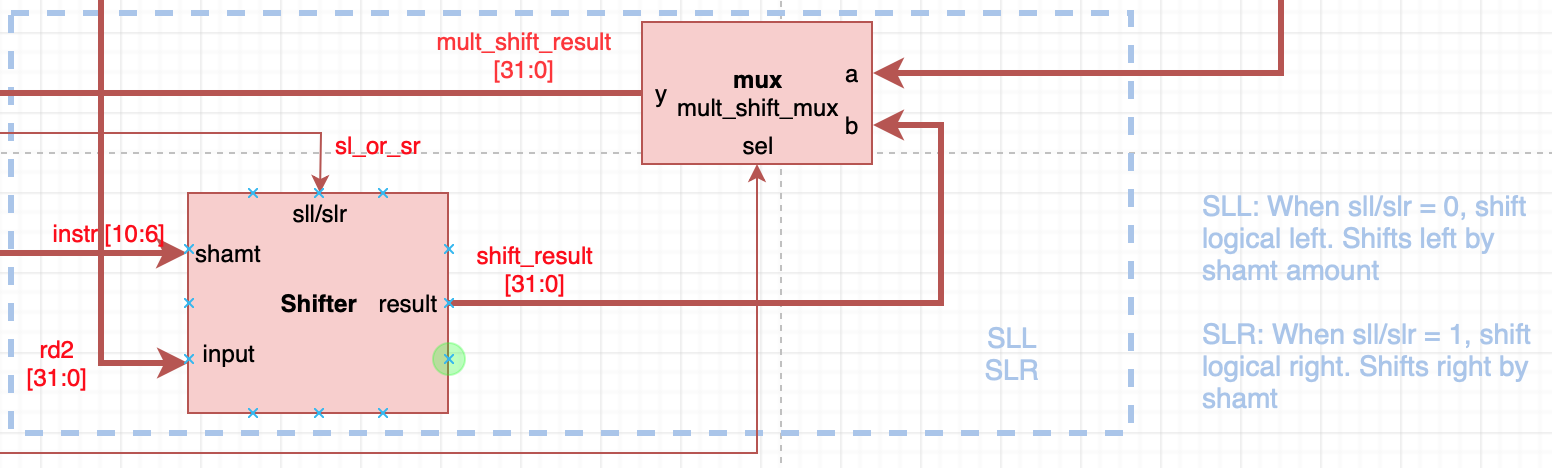


Figure A. SLR and SLL extended design. When sl\_or\_sr is provided a 1, the bits are shifted to the right. When it is provided a 0, the bits are shifted to the left.

**JR**:

The JR instruction allows program execution to jump to an address from a register provided by a parameter to the JR instruction. That is, JR gives a register address to retrieve an address pointing to a different instruction in instruction memory. To enable this functionality, JR must be able to get data from the register file and move that address into the program counter data register. Since JR is an R-Type instruction, the register to retrieve data from is given in the register source field of the machine code. Therefore, minimal work is needed to support this instruction as any R-Type instruction feeds this field (bits 25-21 of an instruction) into a read address port of the register file. We created a multiplexor (mux) that takes the input from the jump mux from Assignment 5 and another input from read data port 1 of the register file. The jump mux selects either program counter + 4 or a “jump-to address”. Therefore, this newly added mux labeled “pc\_jr\_mux” must select between the output of the jump mux or the data from read data port 1 of the register file depending on the instruction being executed. Data selection of this port is controlled by a new pc\_jr\_sel control signal provided by the control unit. The control unit supplies a 1 for this signal when a JR instruction is executed, which selects port 1 of the pc\_jr\_mux. The output of the pc\_jr\_mux is fed to the program counter data register and clocked in on the next clock cycle. The design of this mux is shown in Figure B below.

As an aside, we struggled with the design of the JR instruction because we fundamentally misunderstood what the instruction was doing. We believed that the JR instruction stores an address to $ra which is actually what JAL does.

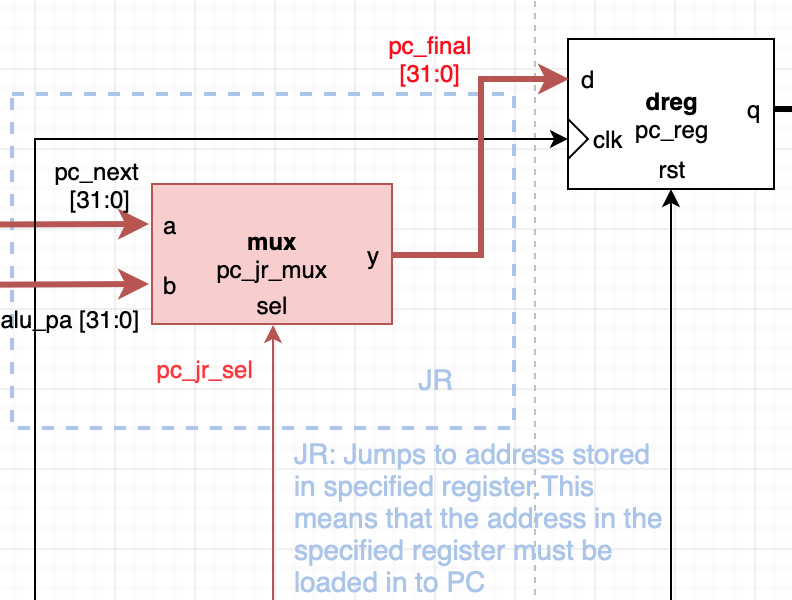


Figure B. JR extended design. When pc\_jr\_sel is provided a 0, the pc\_next value is provided to the mux and sent to the register. When pc\_jr\_sel is provided a 1 the jr\_addr is provided to the mux and sent to the register.

**JAL**:

The JAL instruction was the most difficult for us to implement because of our confusion with what JR and JAL does. JAL is an instruction used to jump to a different portion of a program while storing the return address of the next instruction (program counter + 4) into register 31, otherwise known as $ra. JAL also loads the program counter register with the address of the jump target in the assembly program. Therefore, to implement this, we first turned to the provided MIPS Reference Data sheet. Because JAL is a J-Type instruction, we knew the jump address to be loaded into the program counter register was as follows: { PC+4[31:28], address, 2’b0 } where address is given by the address field in the machine code. Thankfully, this concatenation of bits was already done for us in Assignment 5 to support the jump instruction. The data bus carrying this jump address was already connected to pc\_jmp\_mux from Assignment 5, then to the new pc\_jr\_mux, and finally to the program counter register (pc\_reg). Therefore, when a JAL instruction is executed, the control unit outputs signals to select the correct input ports on these muxes.

Next, we needed to add support for writing to the $ra (register 31) register in the register file whenever a JAL instruction was executed. To do this, we added another mux, rf\_wa\_jal\_mux, connected to the input write address port of the register file. This mux also takes input from the rf\_wa\_mux given in Assignment 5 (which picks between bits 20-16 or bits 15-11 of the input instruction as the write address for the register file) and a hard-coded 5 bit value for 31. The input control signal rf\_wa\_jal\_sel selects the hard-coded 5 bit value as an input to the register file when JAL is executed. This covers writing directly to $ra despite the machine code for JAL having no register address to right to.

Finally, we needed to support storing the address of the instruction after JAL to $ra. The next instruction is otherwise known as program counter + 4. Since an adder from Assignment 5 already outputs this pc\_plus\_4, we can take this data bus and feed it to another new mux, rf\_wd\_jal\_mux. This mux takes the pc\_plus\_4 value as input and the output of an old mux, rf\_wd\_mux which selects between data from data memory and the output of the ALU. A new control signal, rf\_wa\_jal\_sel, picks pc\_plus\_4 as the output data when JAL is executed. The output of this new mux rf\_wd\_jal\_mux is not directly connected to the write data port of the register file as there are other sources of data to consider when writing to the register file: the output of a MFLO/MFHI or SLL/SLR instruction. Therefore, another mux exists to select between the output of rf\_wd\_jal\_mux and those instructions. Figure C shows our implementation of the JAL instruction. Please see Appendix A for the full integration of this instruction along with the jump address from Assignment 5.

All control unit signals are provided on the basis that these are J-Type instructions with different op-codes. Therefore, the control unit logic can distinguish between when to output high for certain signals like rf\_wa\_jal\_sel.

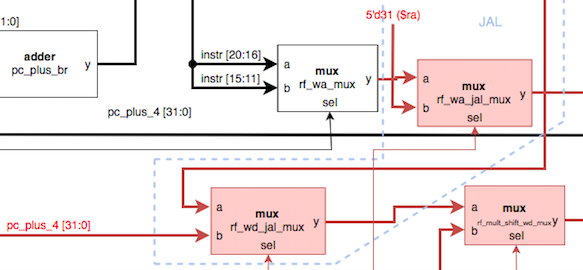


Figure C. JAL MIPs extended design. When rf\_wa\_jal\_sel is provided a 0, the value from rf\_wa\_mux is provided for writing the address to the register file. When rf\_wa\_jal\_sel is provided a 1, the value stored in the $ra register is provided for writing the address to the register file.

**MULTU**:

Our original plan for the MULTU instruction, which multiplies two 32 bit values together to produce a 64 bit output value, was to use the ALU to perform multiplication. We were advised against this as the ALU would take too long to accomplish this instruction.

Therefore, our MULTU implementation makes use of a new multiplication module which takes two 32-bit values from the first two read data ports of the register file. The logic for this module is simple: A x B = Y. There are no control signal necessary for this module. The multiplication module (MULT) outputs its 64 bit value to two registers: HI and LO. Because our processor only supports 32 bits at a time, these two registers were necessary to divide the output of the MULTU instruction. The upper 32 bits of the output are stored in HI and the lower 32 bits are stored in LO. Figure D below shows the new MULT module and how its connected to the HI and LO registers.

**MFLO and MFHI:**

To actually retrieve data from the MULTU instruction, MIPS includes the MFHI and MFLO instructions. These retrieve the upper or lower 32 bits of the multiplication result and stored it in a specified register in the register file. To support these instructions, we added the HI and LO registers as described in the MULTU instruction. These registers also have a write enable input and a clock input to function like a normal register. Data is only written to these registers when write enable is high. This control signal, mult\_we, is provided by the control unit whenever a MULTU instruction is executed. The address of the register to write data to is given in the register destination field of the machine code for these instructions. Therefore, rf\_wa\_mux selects bits 15-11 of the instruction with a low reg\_dst control signal and our new rf\_wa\_jal\_mux forwards this address to the register file with a low rf\_wa\_jal\_sel signal.

The outputs of the HI and LO register are fed to a new mux mult\_hilo\_sel that picks which register to forward data from depending on the instruction being executed; that is, MFHI tells the control unit to give a low mult\_hilo\_sel signal to pick the data from the HI register, and MFLO tells the control unit to give a high mult\_hilo\_sel signal to pick the data from the LO register. Data is then output to another mux, mult\_shift\_mux, as described in the SLL and SLR section. This mux picks between the output of the new shifter and the output of the mult\_hilo\_mux using a new mult\_shift\_select signal. Simply, a MFLO/MFHI signal tells the control unit to output a 0 for this signal to select data from the LO/HI register, and a SLL/SLR tells the control unit to output a 1 for this signal to select data from the shifter. The control unit determines all this from the function field of these instructions as MFHI/MFLO/SLL/SLR are R-Type instructions with the same op-code.

Figure D below shows the new modules to support MFHI and MFLO. Please see Appendix A for integration with the rest of the datapath.

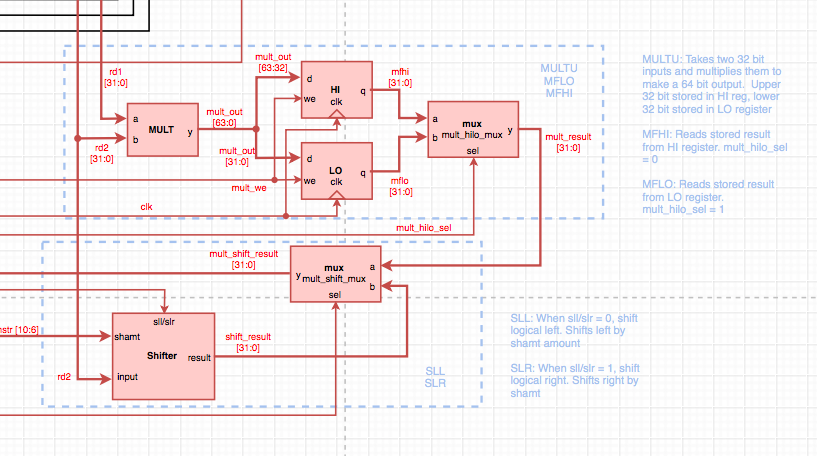


Figure D. MULTU, MFLO, and MFHI extended design. When mult\_hilo\_sel is provided a 0, the contents of the LO register are selected. When mult\_hilo\_sel is provided a 1, the contents of the HI register are selected.

Table 1. Function of modules

|  |  |  |
| --- | --- | --- |
| **MIPS instruction** | **Module(s)** | **Functionality** |
| SLR | Shifter | Shifts an input value to the right a specified number of bits. |
| SLL | Shifter | Shifts an input value to the left a specified number of bits. |
| JR | pc\_jr\_mux | Updates the program counter with the value of the jump target address. Jump to the target address. |
| MULTU | MULT, HI, LO | Multiplies two unsigned integers. |
| MFLO | LO, mult\_hilo\_mux | Retrieves the lower 32 bits of the multiplication instruction and stores the value into a register. |
| MFHI | HI, mult\_hilo\_mux | Retrieves the upper 32 bits of the multiplication instruction and stores the value into a register. |
| JAL | rf\_wd\_jal\_mux, rf\_wa\_jal\_mux,  rf\_mult\_shift\_wd\_mux | Jumps to the target address location after storing a return address by incrementing PC +4 to reach the next line. |

**Accomplished Tasks**

1. *Control Unit of MIPS microarchitecture*
2. *Microarchitecture design which includes the following extended MIPS instructions:*
   1. *SLR and SLL,*
   2. *JAL and JR,*
   3. *MULTU, MFHI, and MFLO*

**Conclusion**

The tasks of the lab were accomplished successfully. The SLR, JAL, MULTU, SLL, MFHI, JR, and MFLO MIPS instructions were extended in the microarchitecture design. Although we struggle with implementing a few of the instructions, we were able to receive guidance from our lab TA and peers. The CU truth table was developed to provide the signals for MIPS extended instructions. Rather than verifying each value at a time, we learned that 0’s can instead be implemented for simplification of testing purposes instead of “don’t-cares”. The CU output table values correspond to the expected input and outputs of the components which support the MIPS extended instructions. We were able to further develop a foundation of microarchitecture design and learned how to extend MIPS instructions for further enhancement of a single-cycle processor. Assignment 7 Part 1 provided preparation for the hardware validation phase of extended MIPS instructions upon the FPGA board.

**Appendix A: Extended MIPS Microarchitecture**

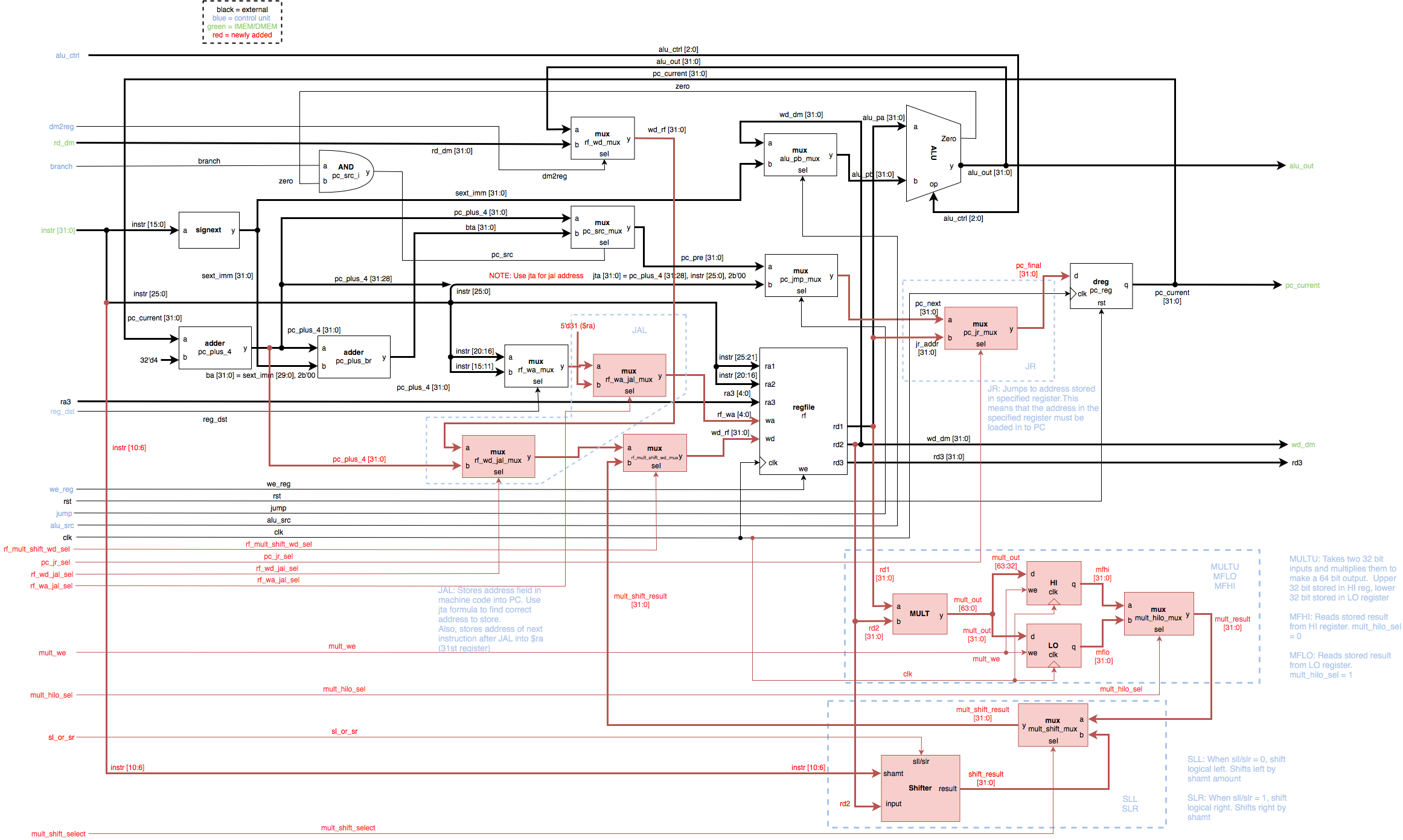
****

Figure F. First half of the microarchitecture design

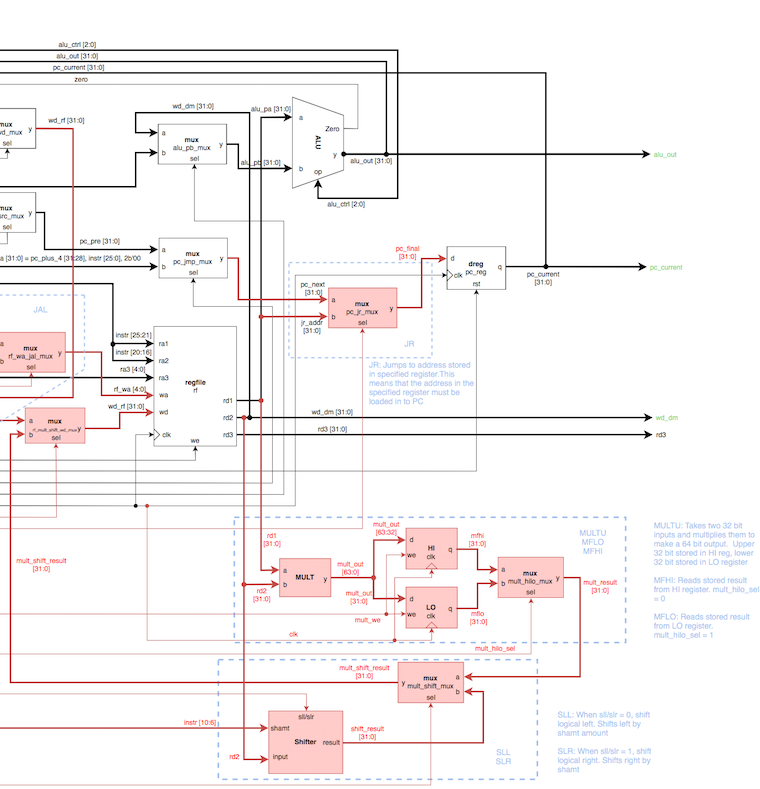
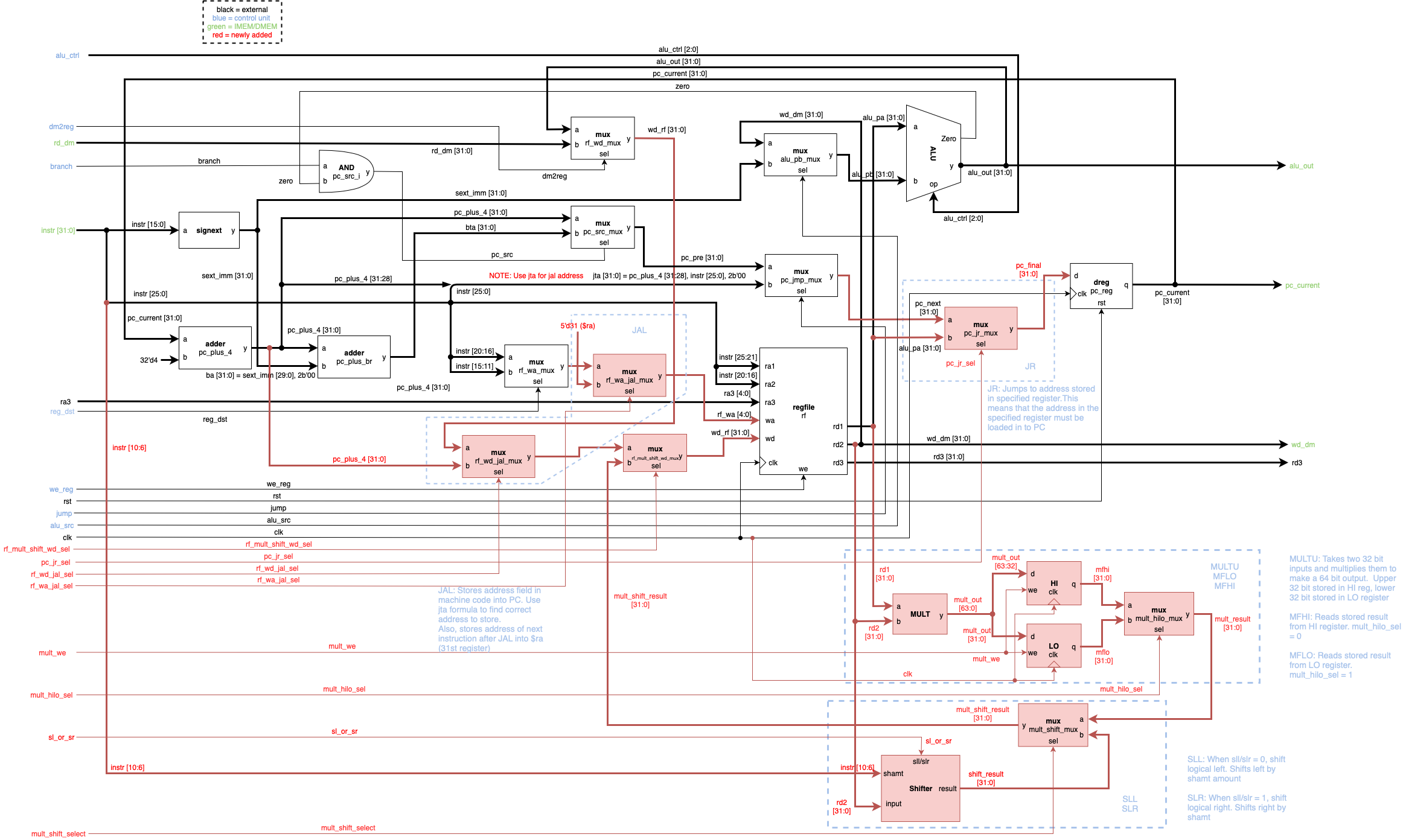


Figure G. Second half of the microarchitecture design



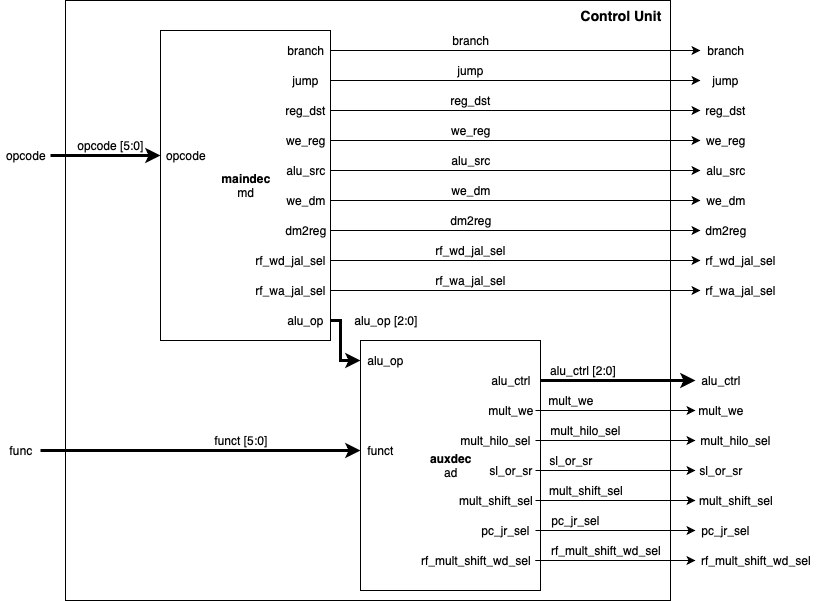
****

Figure H. Control unit architecture

**Appendix B: Control Unit Truth Table**

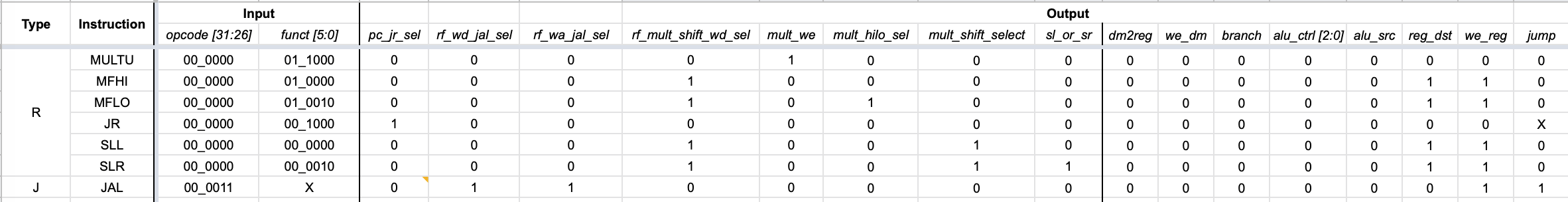
****

Figure I. Control unit truth table for extended MIPS instruction