ELEC2602: Processor Project Report

Matthew, McAuliffe	Liam, Kerr	Adrian, Le
530819475	530504375	530147596

May 2025

1 Hierarchical Datapath Diagram

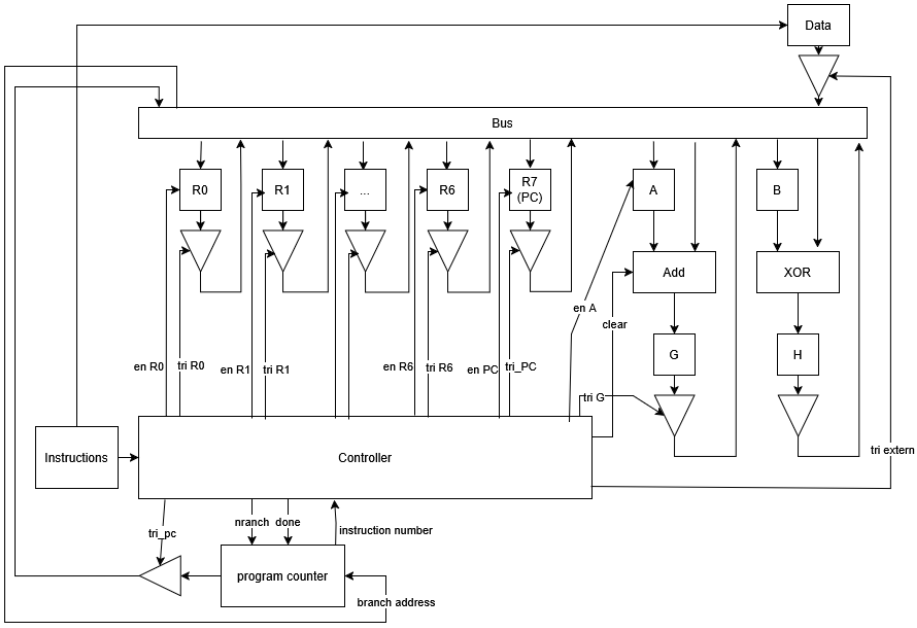

Figure 1: Datapath of CPU

Registers We have 8 registers each of 16 bits. One of these serves as a dedicated register to hold the program counter value for branching. Our controller outputs 2 registers, a TRI REGISTER, which controls writing to the bus, which is one-hot enforced to prevent clashing. And an ENABLE REGISTER, which allows for reading from the bus. The lower 8 bits of each are for the primary registers, with the higher bits reserved for ALU registers, data, and program counter.
$en - reg[11:0] = a, g, b, h, (primaryregisters)$

$tri - reg[11:0] = PC, g, h, data, (primaryregisters)$

ALUs Our ALU's each have a dedicated input and output register. We have implemented both an adder and a bitwise xor. The code for an equality comparator is present, but the module is not connected. Its structure is the same as the existing ALUs, except that its output is a flag sent to the controller, such that we can use it as a branch condition.

Other than our registers, we have 2 special inputs to the bus, out data input, allows us to load external data onto the bus. Our program counter can also read and write to the bus such that we can save a PC value to return to, or load a value to jump to.

Our program counter allows for up to 64 instructions (6 bits).

2 Finite State Machine

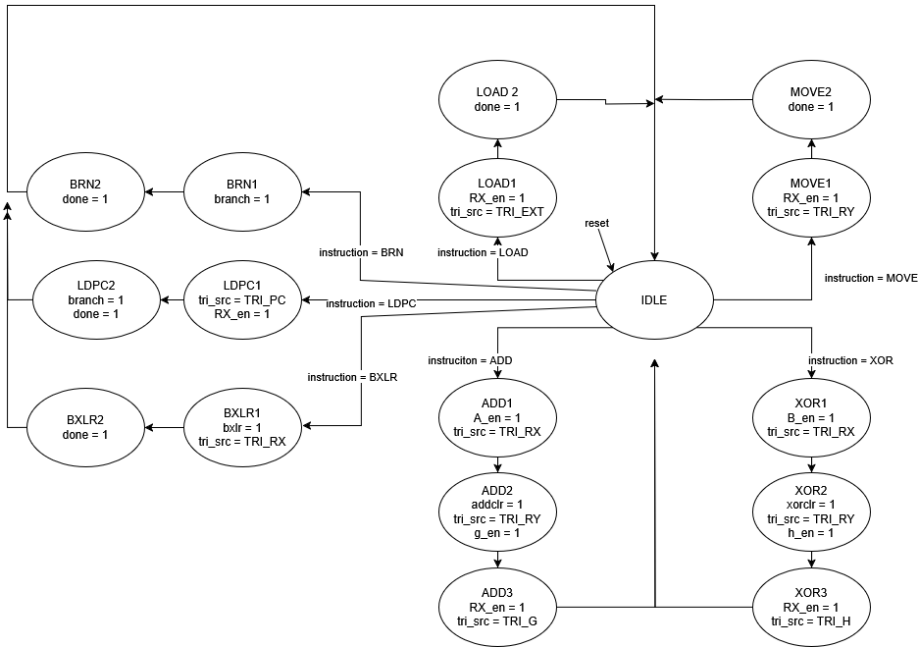

Figure 2: Controller State Machine (see notes below)

NOTE: Assume all output values are 0 unless specified.
NOTE: Not shown due to number of arrows, but the final stage of each instruction can go to the first of the next instruction if present. ie, IDLE can be skipped if there are more instructions.

Instructions

- **Load**: Enables data to write to the bus, and enables the specified register to read from it.
- **Move**: Works the same as load, but a register writes to the bus rather than data.
- **AND and XOR**: Load each value into the ALU separately before performing the operation and writing that value back onto the bus and into the specified register.
- **LDPC**: LDPC1 loads the current value of the program counter into the dedicated PC register for storage and retrieval later, LDPC2 raises the branch flag to overwrite the next instruction address.
- **BRN**: Raises the branch flag to allow for a new branch address to overwrite the program counter address.
- **BXLR**: Retrieves the stored program counter value in the PC register, and overwrites the program counter with this value, this allows us to return from the function call.

All instructions have their own dedicated loops from IDLE. Instructions that could be completed in 1 clock cycle are now completed in 2, as an extra state has been added to output the done flag to increment the program counter.

A typical use of the branching would be;
LDPC to save the current location, $->$ BXLR to return once the function is complete, $->$ BRN to branch to a new function.

All final states in each instruction output a DONE signal to increment the program counter, except for BXLR2. BXLR2 serves as a buffer state, so that the branching works correctly, incrementing the program counter here would result in the wrong instruction being branched to.

Due to issues with how state machines work in modelsim, we had to take the previous instruction rather than the current one, this does not affect the output, but does delay it.

Our ROM is loaded with the following instructions

1. **LOAD**: Load bus value to R1.
2. **LOAD**: Load bus value to R2.
3. **ADD**: Add R1 and R2, store sum in R1.
4. **LDPC**: Load PC value to PC register.

5. **LOAD**: Load bus value to R1.
6. **MOVE**: Move value stored in R1 to R3.
7. **BXLR**: Return to address stored in PC register during step 3.
8. **BRN**: This step branches back to first instruction.