IPA Project Report

Mid-Eval Progress

Ananya Sane (2020102007) L Lakshmanan (2020112024)

Overview

Up to this point, we have written code for all 5 stages of a sequential processor SEQ. Each stage is implemented in its own module, with one final module `processor.v` to instantiate each block and another module for a centralised register array with read and write operations called `regarr.v`, connect the outputs of one stage to the input of the next, and create the final processor. In our implementation, all 5 stages happen in one clock cycle as is required for the sequential implementation, with the PC being updated to the newPC value at every positive edge of the clock.

The processor has 2kB of instruction memory, 14 registers and 2kB of data memory.

Implemented Instructions

- `halt`
- `nop`
- `cmovXX`
- `irmovq`
- `rmmovq`
- `mrmovq`
- `OPq`
- `jXX`
- `call`
- `ret`
- `pushq`
- `popq`

Stage 1: Fetch

The fetch stage works on the instruction memory `insmem`, reading 10 bytes at a time. `icode` and `ifun` are split and aligned from the first byte, and valP is decided based on the value of icode. From the second byte of the fetched instruction, the register operand specifiers are also obtained and stored. In the case where the instruction is 10 bytes long, the eight byte constant is stored in valC.

Thus, the inputs and outputs to this stage are as follows:

Inputs:

- `clk`
- `PC`

Outputs:

- `icode`
- `ifun`
- `rA`
- `rB`
- `valC`
- `valP`
- Status conditions:
  - `inst_valid` : set when inst is valid
  - `imem_er` : set when address is invalid
  - `hlt_er` : set when halt is encountered

Other Paramaters

- `insmem` : register array that functions as the instruction memory
- `inst`: register that is used to fetch 10 bytes from `insmem` at the location pointed to by PC

The instructions are hardcoded into the processor in this stage in an `initial` block for now. We aim to be able to read instructions from a file into `insmem`.

Stage 2: Decode

In this stage, the instruction is decoded from the `icode` value and the required values (usually `valA` and `valB`) are obtained from the registers `rA` and `rB` which are read from the central register bank according to operand specifiers that were obtained from the fetch stage. The stack pointer is also required for a few of these instructions.

Inputs

- `clk`
- `icode`
- `valAin`
- `valBin`
- `stkPt`

Outputs

- `valAout`
- `valBout`

Stage 3: Execute

The ALU is instantiated in this stage, and the results of computations on `valA` and `valB` are stored in `valE` (where applicable). In most cases, this is an `OPq` instruction from the ALU. Also, the three flags used by this architecture: zf, of and sf are computed in this stage. The flags are set for the `OPq` instructions and are used for the conditional instructions.

Inputs

- `clk`
- `icode`
- `ifun`
- `valA`
- `valB`
- `valC`

Outputs

- `valE`
- Condition Codes:
  - `cnd`
  - `of`
  - `zf`
  - `sf`

Note: The ALU module has absolute paths for including the various sub modules. They will have to be modified accordingly.

Stage 4: Memory

The portion of instructions that require the altering of or reading from data memory is done in this stage. It is here that we interact with the actual memory of the device with read and write operations.

Inputs

- `clk`
- `icode`
- `valA`
- `valB`
- `valE`
- `valP`

Outputs

- `valM`

Other Parameters

- `datamem`

Stage 5: Write Back

Writes either `valE` or `valM` to the required registers in the instructions that call for it. Therefore, this stage handles register updates.

Inputs

- `clk`
- `cnd` (condition code for cmovXX)
- `icode`
- `rA`
- `rB`
- `valM`
- `valE`

Outputs

- `dstA`
- `dstB`
- `dataA`
- `dataB`

Stage 6: PC Update

Sets `newPC` value to `valP` in non conditional instructions, and to `valP`, `valC` or `valM` in instructions that invoke control transfer depending on whether the condition is met.

Inputs

- `clk`
- `cnd`
- `icode`
- `valP`
- `valM`
- `valC`
- `PC`

Outputs

- `newPC`

Processor Wrapper

The `processor.v` file includes all the required files for all the stages as well as the register bank. This code is mainly meant to set the status conditions for the processor and to monitor and end execution if necessary. The clock is also controlled by this code with an always statement.

This module takes no arguments as all the required files are included and the modules are instantiated. The value of `PC` is updated in an always block here at the positive edge of the clock, hence beginning execution of the various stages.

Register Array

This module stores the data contained in all 14 registers and stack pointer using a two dimensional array `regArr`. It is capable of reading and writing to this array of registers depending on the input specifiers given, maximum being two at a time. It takes in the specifiers for the required registers and values (if applicable) to be written to them, and performs the required operations accordingly.

Inputs

- `PC`
- `rA`, input for the read operation
- `rB`, input for the read operation
- `dstA`, input address for write operation
- `dstB`, input address for write operation
- `wrtA`, input values to be written
- `wrtB`, input values to be written

Outputs

- `valA`, value in the register specified by `rA`
- `valB`, value in the register specified by `rB`
- `valStk`, value of the stack pointer `%rsp`

Results So Far

The code for the processor compiles without any errors, and the executable file `a.out` runs without issues according to the instruction memory given. The output of the instructions given to the processor and the program state has been given in the `output.txt` file that has also been uploaded to the repository.
