ELEC2204 – Computer Simulation

Date: 27/03/2025   
Leah Tiyane Nyahunzvi, 35357045

Email Address: [ln2u23@soton.ac.uk](mailto:ln2u23@soton.ac.uk)

*This report follows details the design and implementation of a five-stage pipelined processor. The basis of the design is the MIPS architecture and instruction set, and it is emulated in C. The final design takes an assembly.txt file and converts it to a binary.txt, which the processor reads line by line and feeds into the pipeline.*

**1. Design**

**1.1 Processor Design**

This is a 32-bit pipelined processor with five stages, namely, Fetch, Decode, Execute, Memory Access, and Write Back, in that order. There are 31 general purpose registers used during program execution, meant to mimic stack memory, and four pipeline registers, one between each stage. Each instruction is passed fully along the pipeline, so that forwarding and stalling requirements can be calculated at each stage.

* 1. **Processor Layout**

The design detailed above is illustrated in figure 1  
A diagram of a machine

AI-generated content may be incorrect.

Figure 1: Processor Layout

**1.2 Registers and Memory Layout**

As mentioned, there are 31 general purpose registers, t1:t31, in line with the 5-bit operands. The first register, t0, is a special purpose register and always holds zero. This is useful for initialising registers at the beginning of programs. The main memory is 32 bits wide and has 131071 addresses, due to the maximum 17 bits which can be referenced in ‘load’ and ‘store’ instructions. The program memory holds one instruction at a time, retrieved from a text file.

A graph of memory and a few words

AI-generated content may be incorrect.

Figure 2: Memory Layout

**1.3 Instruction Set Architecture**

A screenshot of a computer code

AI-generated content may be incorrect.

Figure 3: Instruction set with example uses

The full instruction set with syntactical examples is visible in figure 3.

**A close-up of a colorful box

AI-generated content may be incorrect.**

Figure 4: Diagram showing instruction formats: R Type, I Type, J Type

As illustrated in figure 4, all opcodes are 5 bits wide, and the two leftmost bits determine whether the instruction is I-type, R-type or J-type. Opcodes in the form of 01xxx and 11xxx are I type instructions, where one operand is a constant, while opcodes in the form 10xxx are R type instructions, where all operands are register addresses. Jump and NOP instructions are the only ones to begin with 00xxx. Jump instructions contain only an opcode and a jump address 27 bits wide, allowing a user to jump the entirety of a program with up to 2^17 -1 = 134 217 727 instructions. Relative branches have not been implemented; however, the emulator has been written in such a way that the IS can be easily expanded with the remaining unused opcodes.  
R type instructions use 5 bits for rs, rt, and rd. I type instructions use 5 bits for rs and rt, and 17 bits for the immediate value. J type instructions only need 5 bits for the op code and the remaining 27 are for the address.

**2. Functionality**

The final design relies heavily on C structs, with attributes labelled, ‘now’ and ‘next’ used to simulate sequential behaviour. This was deemed to be the most effective method, as a single update function updates all “now” values with their “next” counterparts. It also ensures that all the data is settled by the control logic before the next pseudo clock edge. Components and elements of the processor have been declared as global variables, accessible at any point in the program, and each pipeline stage is a function that can edit and update the processor elements. The main program calls the functions fetch, decode, execute, check\_hazard, memoryaccess, writeback and update in an infinite loop until there are no more instructions to be read from the file, at which point, the processor feeds bubbles into the pipeline to allow the final instruction to make it to the writeback stage. Each stage in the pipeline operates on “now” values, while setting up “next” values which will take effect after the system updates.

**2.1 Pipeline Stages**

**2.1.1 Fetch**This function opens the text file containing the string of binary instructions and reads it line by line until it locates the line the program counter is pointing to, at which point, it retrieves it and converts it into a 32-bit unsigned integer. The program counter is then incremented and points to the next instruction before the file is closed. Once the instruction has been fetched and converted, the value is fed to the input of the fetch/decode pipeline register.

**2.1.2 Decode**The instruction is then analysed based on its opcode. Common instruction types are typically treated the same, with a few exceptions (example; branches and memory operands). The data to be fed from the alu is then read from register files or passed directly from the immediate operand to the decode/execute pipeline register. At this point, the first stage of hazard detection takes place. The forwarding logic checks if the writeback address in the EX/MEM or MEM/WB registers are the same as one of the source addresses for the ALU. If this condition is met, the data is forwarded from the pipeline registers into the ALU. Forwarding from EX/MEM takes precedence over forwarding from MEM/WB, because this will naturally have the most up-to-date data for that register address.

**2.1.3 Execute**Data hazards have been resolved at this point, and the data fed into the ALU is operated on. The result is propagated through to the MEM/WB pipeline register.

**2.1.4 Check Hazard**This is the second stage of hazard detection. Similar to the first stage, data is forwarded from the execute/memory register to the ALU input for the next cycle. It is important to note that store instructions do not trigger forwarding. This function also checks for “load-use” hazards, wherein the data will not be ready in the register file at the time that its requested. If detected, the pipeline stalls until it is resolved.

**2.1.5 Memory Access**This function will only make changes to the status of the processor if the instructions in EX/MEM are load or store instructions. Load instructions retrieve the data at the specified address and pass it through to MEM/WB, while store instructions simply overwrite the memory address with data from the alu.

**2.1.6 Write Back**The status of the processor is only altered if the instruction is R type or specific I type instructions (example; bnq and beq do not write back). The “next” value of the specified general purpose register is assigned, and this is where the pipeline ends. No data or instructions are forwarded beyond this point

**2.2 Build Functionality**

**2.2.1 Assembler**  
The processor also sports an assembler, which converts assembly code in a txt file into binary instructions corresponding to the custom architecture. Information regarding the assembler and syntax rules can be found in “README.txt”, which also details synctactic and semantic rules. The assembler looks for an “assembly.txt” file and analyses it line by line, storing the converted 32-bit instructions into a “binary.txt”. This file path is where the processor looks for instructions during the fetch stage.

**2.2.2 Build**The parser and compiler are run in a single batch file. The build command clears the terminal and deletes and old executables existing in the directory before beginning execution. This considerably reduces the build time while ensuring that any executables in the directory are the most updated version.

**3. Basic Testing**

Short assembly programs were written to test instructions as the processor was developed. The process proved to be iterative, in that when the design was altered to expand functionality and passed the associated test, all previous tests were redone to ensure that all systems continued to work as required regardless of the expansions. The key tests include branch tests, data hazard tests, and the testing of various loops.

**2.1 BNQ Test**A computer code with black text

AI-generated content may be incorrect.

Figure 5:BNQ test algorithm and results

**3.2 BEQ Test**A screenshot of a computer code

AI-generated content may be incorrect.

Figure 6:BEQ Test algorithm

**3.3 Multiplication Testing**A screenshot of a computer

AI-generated content may be incorrect.

Figure 7: Multiplication test algorithm

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 8: Multiplication test results

The above test uses a simplified version of booths algorithm to multiply two positive numbers. This method for multiplication was chosen as the most efficient and effective for the display program. This is because the same amount of loops will be required for each number. It involves a lot of data being moved initially; however the total execution time is considerably small and doesn’t increase with the size of the number being multiplied. As expected, the result of 3x4 is successfully stored in t6.

**4. Showcase Testing**Two algorithms were considered for the display program. The first, saved as “test\_square” in the directory is simple and robust, however it is very inefficient. This algorithm simply adds a number n to itself n times, and as n gets larger, the amount of time required to perform the function grows exponentially. This algorithm was satisfactory for testing, however, the processor required an excess of five minutes to reach the mid sixties in the execution. As a result of the extreme time cost, this algorithm was not considered for the final showcase.  
  
The chosen algorithm is the “shift and add” algorithm, derived from Booth’s multiplication algorithm. Each number only requires 8 loops to find its square, and while it is unnecessary for smaller numbers, this is quickly made up for as the numbers get larger. In this program, t8 holds the upper compare value, such that the program will terminate after t9 exceeds 200. We expect t6 to take the values of the squares throughout the execution, and the results to be stored sequentially in memory. These results are highlighted below.

**4.1 Showcase Algorithm and Results**A screenshot of a computer program

AI-generated content may be incorrect.

Figure 9: Showcase algorithm using shift and add multiplication

A screen shot of a computer

AI-generated content may be incorrect.

Figure 10: Showcase results

**5. Conclusion**

This is a well-designed pipelined processor that accurately performs each stage pf the pipeline, however, as with all designs, it has room for improvement. The cycle period could potentially be increased if the program text file was only opened and closed once throughout the program execution. Furthermore, R type instructions don’t utilise the bottom seventeen bits of the instruction, resulting in a waste of resources. The instruction set may need to be expanded to minimise this wastage. Furthermore, the parser could be improved to allow a branch point to be referenced before it is defined, as this will make the programming process much easier for the user.

**Appendix**

1. Further information: Readme.txt