Project Final Report
Three Stacks Architecture
Team Red

Allen Liu

Elvis Morales Campoverde

Joye Ma

Xingheng Lin

Rose-Hulman Institute of Technology

Table of Contents

1. Introduction

Our processor uses a Three Stack Architecture to store data into three stacks in memory instead of storing data into registers. The stacks used in our design are: a working stack that holds inputs for the program, a data stack the holds the arguments and return values of a procedure, and a return stack that holds the return addresses. We also included a flag register that helped to determine the condition of the branch. The interrupt and exception functionality fulfilled to provide higher usability for users.

2. Overview

2.1. Instruction Set Design

We use two types of instructions. O type only contains 6-bits opcode without any immediate field. I type have both 6-bits opcode and 10 bits immediate fields. For the datapath, both O type and I type are treated the same because instruction is 16 bits, and the rest of the 10 bits for O type are ignored.

2.2. Implementation

2.2.1. Memory Part:

The memory part included the distributed memory block, the pc, and a couple of multiplexers. The multiplexers selected: the source that would be put into pc, where things would be placed in memory, and what would be written into memory. The memory part would be connected to all the other parts. It would receive inputs for the pc from the ALU part. Inputs on where things would be placed from the Stack pointer part. Inputs on what to put into memory would come from both the Fetch and the ALU part

2.2.2. ALU Part:

In the ALU implementation, we designed our ALU to support six operations: “add”, “subtract”, “or”, “and”, “left shift”, and “right shift”. We implemented the “add” operation by using a built-in adder of the board. For the “or” and “and” operation, we use the “or” and “and” gate to implement those operations. For the “left shift” and “right shift” parts, we implemented with the 16-bit multiplexer. In the ALU output port, we put a 16-bit register to store the ALU operation result. And before the two input ports for the ALU operands, we put two multiplexers to select the operands taken by ALU at this cycle. Below the ALU, we put a 16-bit register to hold the value of the flags triggered by operation from ALU. And below the flag register, we put a compare state register to hold the result of comparing instruction used for conditional branching.

2.2.3. Fetch Part:

In the fetch part implementation, components 1-bit shifter, a 10-bit shifter, and extender are firstly built and tested. Two temporary registers $A and $B are built with reset function. A 16-bits input demultiplexer is used to pass the memory output to $A, $B, $IR, or $OTR. All these components are combined into one piece of fetch part.

2.2.4. Stack pointers Part:

This component consists of three registers, a decoder, and a bunch of multiplexers. A multiplexer is placed in front of each register's input bus to control if the register stores a default value or a value coming from another component. A decoder is used to choose which register the input data is written to. A multiplexer at the output of all registers manages which stack pointer data to output.

2.2.5. Exception Part

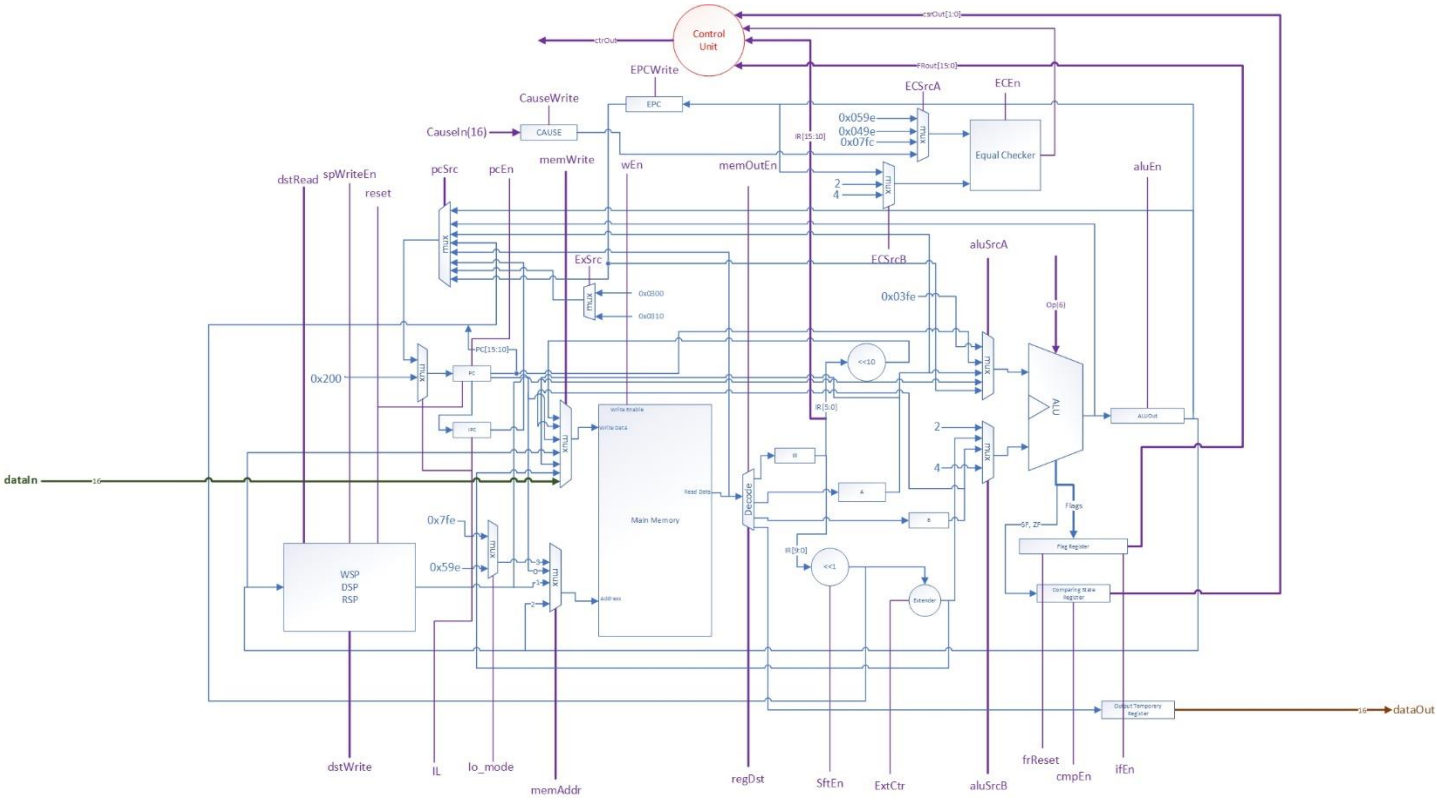
The exception handling involves two new registers and an equal checker. One register is EPC, which stores the address of the instruction at where the exception occurs. The other is CAUSE, storing the cause code of the exception. The equal checker consists of a set of XOR gates to compare if the two input values are equal. Two multiplexers are placed before the input wires of the checker to control input sources. The equal checker is used to check if the stack pointer has reached the set to that specific stack. The control is responsible for processing the input signals and deciding if it has to send the program into the exception handler.

2.2.6. Interrupt Part

When implementing interrupt exception, we added an $ipc (interrupt program counter register) to store the value of the address of the instruction to be executed when the interrupt handler returns to the main program. And when the interrupt is triggered, the control unit will select the 0x0200 and write that into the $pc, and $ipc will be taking the current output of the $pc, which backs up the corresponding $pc + 2 into it. And the next instruction to be executed will be the instruction in the first line of the interrupt handler.

2.3. Final Model

Four parts from implementation combined to make the main body of the model.



3. Unique features

3.1. Assembler

The assembler was written in Java and allows to user upload to a file containing their assembly code. The assembly code is first parsed to remove comments and white spaces. The labels used in the assembly code are recorded, the labels removed, and references to labels within instructions are replaced with the appropriate immediate value. During the parsing step, the number of assembly code lines is checked to assure that the user did not exceed the allowed number of instructions. Due to memory constraints, we only allow the user to write a program that is 256 lines long.

Once the parsing process is complete, the parsed assembly code lines are translated line by line. The instructions are translated to their appropriate opcode. The immediate value of the instruction is converted to binary and appended to the opcode. During the conversion step, the instruction, the length of instructions, and the immediate value are checked to verify that they are valid assembly codes. The assembler allows the user to write immediate values in decimal, binary and hexadecimal.

Once the assembly code is translated into machine code, the translations are written into three files: a .out file that contains the raw translation, a .coe that can be used to generate a new core when generating a new memory block, and a .mif that changes the data stored in a memory block.

The assembler was tested against our manually translated RelPrime program to assure the assembler was translating code correctly. The various smaller unit test was written to make sure that the assembler could handle potential issues.

3.2. Stack Buffer

Stack buffer is a word located at the top of memory. It is reserved for external input, such as a spartan 3e board. The stack buffer data can be loaded to the working stack for further usage in the program.

3.3. Special Register

3.3.1. Flag Register

A flag register is used to check the ALU result, which can be later used as the condition for branch or exception.

3.3.2. Cause Register

Cause register is used for storing the cause of the exception when executing an instruction caused exception; the exception handler uses it to determine which decision to make.

3.3.3. Compare State Register

Compare state register is used for storing the value of zero flags and sign flag after cmp (compare) instruction, the result of comparing top two values stored in the top of the working stack.

3.4. Procedure call

A procedure call is easy to manage with three stacks architectures. The return stack is dedicated to storing the return address of callers. Using this feature, we can decide the maximum number of recursions for our processor. The caller uses the data stack to prepare the input arguments to callees in the data stack. The return value can be stored in the data stack by callee after callee finished execution. With the data stack design, the caller can prepare inputs and callees to return any outputs.

The caller starts the procedure call. The caller will first set up arguments by pushing arguments into the data stack, return address into the return stack frame. Callee program is then called, and the return address is saved in the return stack before callee's execution.

The callee will first retrieve the input parameters from the data stack and execute the body program. When the callee finishes executing the program's body, the callee is responsible for restoring the data stack pointer. The callee puts the return value into the block just below the data stack pointer, returns to the caller with the return address stored in the return stack, and finally restores the return stack pointer.

After returning to the caller, the caller will obtain the value returned by callee from the block below the data stack pointer

3.5. Robustness of Hardware Test

Datapath was split into four subcomponents: the ALU with flag register, the main memory with mux, fetch parts including shifters and extenders, and stack pointers part. Every component inside the subcomponent was tested thoroughly using an exhaustive test that including every possible input combination. Subcomponents were built using the tested individual components. The later unit test was run against each subcomponent is then further tested using exhaustive testing. The Overall Datapath is tested by simulating instruction execution without the control. The testing of the overall Datapath cover every path or wire in the system.

3.6. Robustness of Software Test

3.6.1. Interrupt test

After we tested that the entire processor functions as expected, we put an instruction that always jumps back to that instruction during the simulation, which functions as an infinite loop. And after few cycles, we simulate the signal of interrupt input and simulate to test if the current instruction will jump to interrupt handler and return to the next instruction after the interrupt handler finished executing.

3.6.2. Exception test

We added instructions to certain output information into the handling code for each exception in our exception handler. In our test program, we intentionally wrote instructions that would cause an exception and looked at the output the program gave out. After checking that the exception can be successfully invoked, we would observe the program's status. If the exception is ignored, we would have to ensure that the PC's value is set to the following line of the original exception-causing instruction. If the execution must be aborted, we check whether all signals are set to 0.

3.7. Exception Handler

3.7.1. Arithmetic Overflow

When arithmetic overflow happened, the overflow bit in the flag register is set. When the control unit receives a HIGH value of the overflow flag, it sets PC to the address of the exception handler. The exception handler will decide which type is the exception. For handling Arithmetic Overflow, we just ignore this exception and return to the next instruction of the address stored in EPC.

3.7.2. Stack Pointer Overflow

The working stack, data stack, and return stack have the allocated space. If any of the stack pointers exceeds its limit, the stack pointer overflow exception happened. To check the overflow, an equal checker is used. Once the control unit receives an equal signal from the checker, it leads the program to the exception handler. We handle this exception by aborting the program.

3.8. Interrupts Handler

So far in our processor, we only support input in our interrupt handler, that when the program jumps to the interrupt handler and backs up the address of next instruction in ipc, and after the interrupt handler finished execution, the pc will be

3.9. Input-Output

3.9.1. Input

When the interrupt handler triggers the processor to get the input, it will read the value from the input port and store it into the top of the main memory, which is the stack buffer.

3.9.2. Output

$OTR (Output Temporary Register) is used for storing the output value from the processor. The output port will always read the value from the $OTR. When the program is finished executing, it will take the result, which is the value stored at the top of the data stack, and store it into the $OTR.

4. Conclusion

Our three-stacks architecture is robust at managing procedure calls. With the exception handler, the processor can handle arithmetic exceptions and stack overflow exceptions critical to a stack architecture. The interrupt is implemented to take the input from an external source such as the spartan board. This allows the user to input values without having to reassemble their program. Besides, our design is also users friendly because each instruction is straightforward and easy to type with only one immediate field or none. Our processor only uses one ALU and a few registers. This makes our processor efficient and easy to customize if needed. Every instruction only carries a single responsibility that reduces the further need to modify existing instruction due to the side effects. Importantly, our processor is open for extension. A new feature can be added to the processor without the need to modify the current design. Overall, our processor is economical, flexible, and powerful.

5. Appendix of the test result

RELPRIME(0x13b0) :**30.04 ms**

Cycle: **632848**

Instruction: **13271**

CPI: 632857/ 132718 = **4.768** cycle per instruction

Cycle time:**47.5 ns**

RELPRIME(0x0906) :**14.95 ms**

Cycle: **314662**

Instruction: **66005**

CPI: 314662/ 66005 = **4.767** cycle per instruction

Cycle time: **47.5 ns**

RELPRIME(0x754e) :**215.92 ms**

Cycle: **4545648**

Instruction: **953154**

CPI: 4545648/ 953154= **4.769** cycle per instruction

Cycle time: **47.5 ns**