OoO RISC-IV Superscalar Pipelined Processor in SystemVerilog

Abbas Bakhshandeh
UCLA ECE Masters Student
UCLA ID# 105208768
M.S. Capstone Project

Introduction

Goal: To build an out of order execution (OoO) Risc-IV superscalar (2-issue) pipelined processor in SystemVerilog.

Features:

- Out of Order (OoO) Execution
  - In order to improve processor efficiency I enabled OoO Execution by using a reservation station, ROB (Re-order Buffer), and a physical register file to execute instructions out of order while preventing data hazards
- Pipelining
  - Pipelining implemented to increase throughput and allow different modules to work on different parts of the processing stages for the instructions concurrently
- Superscalar
  - Two sets of hardware were created to be able to process two instructions throughout the pipeline

Module Descriptions

- The design was modularized
- The modules are:
  - Fetch
  - Decode
  - Rename
  - Dispatch
  - Fire
  - Retire
- There is also a top module which instantiates the modules and interfaces
- Interfaces used in SystemVerilog to maintain relevant signals in an organized manner

Module	Description
Fetch	Every clock cycle, two input instructions are stored from a text file containing hexadecimal values. Each input instruction corresponds to 4 bytes (32 bits), so 8 bytes (64 bits) are fetched each clock cycle.
Decode	Extracts necessary information from the two instructions which are fetched every clock cycle, including the destination register, source registers, immediate values, function values, and control signals such as AluOp, aluSrc, memRead, memWrite, and memtoReg. These values are stored and passed to the next stages in the pipeline for further processing. The 8 possible instructions for this processor are ADDI, ANDI, ADD, SUB, XOR, SRA, LW, and SW.
Rename	Register renaming is enabled in order to implement out-of-order execution. Architectural destination registers are assigned to available physical registers, and architectural source registers are mapped to their assigned physical register names. Doing this prevents data hazards that could arise when completing out-of-order execution. A RAT (Register Alias Table) in addition to a "free pool" structure is utilized to keep track of which architectural registers are mapped to which physical registers. A physical register file is also created to hold physical register 32 bit values.
Dispatch	The information collected in the previous stages - the decoded signals as well as the renamed physical registers - are placed inside a reservation station. The reservation station contains "ready" bits for each (physical) source register in each instruction, as well as a ROB number and a functional unit number. A ROB (Reorder buffer) is created that stores the new physical destination registers, old physical destination registers (before renaming), and a "complete" bit which indicates whether the instruction has been completed. When both source registers are ready (and a functional unit is available), the instruction can be issued to the functional unit and removed from the reservation station. Up to three instructions can be issued to the three functional units at one time, if they are available. Two of the functional units are for R-type and I-type instructions (ALU) and the third function unit is for memory instructions (LW and SW).
Fire	Once the reservation station issues an instruction in the table to a functional unit, it is processed based on the instruction's control signals (which were calculated in the decode stage), the physical register names, and the values in the physical register file. The final results of these calculations are stored in the physical register file (at the correct physical destination register location).
Retire	While the execution of instructions can be completed out-of-order, the retiring (which actually updates the architectural registers and memory) must be done in-order to prevent data hazards. To accomplish this, the ROB is checked to see which instructions have completed (after the fire stage). It will retire instructions which have had all instructions above it (indicated by a program counter) also retired. This will assign the values stored in the physical register file to their corresponding architectural registers or memory. Up to two instructions can be retired each clock cycle. After retiring, they are removed from the ROB and the corresponding physical registers are freed for further usage by future instructions.

Microarchitectures

- Various microarchitectures were utilized within different modules to implement the different features
  - RAT (Register Alias Table)
  - Free Pool
  - Reservation Station
  - ROB (Reorder Buffer)
- Specific details for each block described in the table to the right

Mircroarchitecture Block	Description
RAT (Register Alias Table)	The RAT is used to keep track of which architectural registers have been renamed to which physical registers. The RAT is created in the Rename module.
Free Pool	The free pool is an array containing 64 one bit values. Each one bit value indicates whether the corresponding physical register is occupied or not. For example, if (free pool)[7] == 1'b1 is true, then this means physical register 7 is occupied (and hence should not be assigned to any other architectural register). In this microarchitecture, p0 is therefore reserved and to not be occupied / maintain a value of 0. This is to aid further calculations within the pipeline as well as to maintain symmetry with the architectural registers in RISC-IV, of which x0 is reserved to always keep a value of 0. The free pool is created in the Rename module.
Reservation Station	The reservation station holds all the decoded information for incoming instructions as well as their corresponding renamed physical registers in a 2D table. This table also contains signal bits that indicate whether the (physical) source registers of an instruction are ready, which will allow - assuming a relevant functional unit is also ready - for the instruction to be executed. All relevant control signals and physical registers are stored in this table, which paired with the physical register file allows for the instructions to be executed. The reservation station is first created in the Dispatch module, and subsequently accessed in the Fire module.
ROB (Reorder Buffer)	The reorder buffer keeps track of what order the instructions are originally in, so that they can be retired in-order. While the processor efficiency benefits greatly from out-of-order execution, the ROB ensures that those instructions are still retired in order to prevent data hazards. The reorder buffer is created in the dispatch module and is subsequently accessed and modified in the retire module.

Verification I

- The design was simulated using ModelSim in Quartus
- Various testbench cases were created in order to verify functionality
  - The input to the processor is a text file with hexadecimal values on each line
  - Each value corresponds to one byte, or 8 bits (therefore 4 values is 32 bits, or one instruction)
  - Example set of assembly language instructions and selected input hexadecimal values shown to the right

```
addi x2, x0, 6
addi x3, x0, 15
andi x5, x3, 1
addi x1, x0, 36
add x4, x5, x2
sub x0, x0, x0
addi x9, x2, 23
xor x6, x2, x3
sub x2, x1, x6
sra x7, x3, x0
sra x3, x3, x5
andi x9, x3, 23
```

```
20 93
21 02
22 40
23 00
24 93
25 00
26 22
27 82
28 33
29 40
30 00
31 00
32 33
33 01
34 71
35 04
36 93
37 00
38 31
39 43
40 33
```

Verification II

- For the assembly language instruction example in the previous slide, the calculations are shown in the top image
- The correct final result for the register file for this same example is displayed in ModelSim (shown below the calculations)

```
addi x2, x0, 6   -> x2 = 0 + 0110 = 0110
addi x3, x0, 15    -> x3 = 0 + 1111 = 1111
addi x1, x2, 36   -> x1 = 0110 + 36 = 101010
sw x3, x3, 9   -> 1111 + 1001 = 11000 (store 1111 in memory[11000])
add x4, x3, x2   -> x4 = 1111 + 0110 = 010101
add x0, x0, x0   -> x0 = 0 (by definition)
sw x1, x0, 0   -> 0 + 0 = 0 -> (store 101010 in memory[0])
lw x2, 24, x0   -> x2 = memory[0 + 11000] = memory[11000] = 1111
xor x6, x2, x3   -> x6 = 1111 ^ 1111 = 0
lw x3, xo, 0   -> x3 = memory[0 + 0] = 101010
sub x5, x3, x4   -> 101010 - 10101 = 010101
sra x7, x2, x9   -> x7 = 1111 >> 0 = 1111
andi x9, x3, 23   -> x9 = 101010 & 10111 = 000010
sw x5, x7, 1   -> 1111 + 1 = 10000 (store 10101 in memory[10000])
```

```
Register File 0 (x0): 00000000000000000000000000000000
Register File 1 (x1): 00000000000000000000000000101010
Register File 2 (x2): 00000000000000000000000000001111
Register File 3 (x3): 00000000000000000000000000101010
Register File 4 (x4): 00000000000000000000000000010101
Register File 5 (x5): 00000000000000000000000000010101
Register File 6 (x6): 00000000000000000000000000000000
Register File 7 (x7): 00000000000000000000000000001111
Register File 8 (x8): 00000000000000000000000000000000
Register File 9 (x9): 00000000000000000000000000000010
Register File 10 (x10): 00000000000000000000000000000000
Register File 11 (x11): 00000000000000000000000000000000
Register File 12 (x12): 00000000000000000000000000000000
```

Results

- Area / power utilization and power results obtained
  - Each module separately partitioned in Quartus
  - Modules with most logic generally use most resources as well as power

Fetch	Total Logic Elements: 1 Total Registers: 0 Total Modular Pins: 65 Total Combinational Functions: 1 Logic element usage by number of LUT inputs 4 input functions: 0 3 input functions: 0 ≤2 input functions: 1
Decode	Total Logic Elements: 183 Total Registers: 138 Total Modular Pins: 189 Total Combinational Functions: 101 Logic element usage by number of LUT inputs 4 input functions: 46 3 input functions: 29 ≤2 input functions: 26
Rename	Total Logic Elements: 423 Total Registers: 85 Total Modular Pins: 185 Total Combinational Functions: 412 Logic element usage by number of LUT inputs 4 input functions: 240 3 input functions: 150 ≤2 input functions: 23
Dispatch	Total Logic Elements: 1576 Total Registers: 1400 Total Modular Pins: 1922 Total Combinational Functions: 204 Logic element usage by number of LUT inputs 4 input functions: 179 3 input functions: 2 ≤2 input functions: 13
Fire	Total Logic Elements: 257472 Total Registers: 3212 Total Modular Pins: 4909 Total Combinational Functions: 257472 Logic element usage by number of LUT inputs: 4 input functions: 245651 3 input functions: 9029 ≤2 input functions: 2792
Retire	Total Logic Elements: 33098 Total Registers: 1060 Total Modular Pins: 36293 Total Combinational Functions: 33098 Logic element usage by number of LUT inputs: 4 input functions: 32789 3 input functions: 150 ≤2 input functions: 159

Area / Resource Utilization Results

Fetch	45.76 mW
Decode	46.20 mW
Rename	46.19 mW
Dispatch	46.12 mW
Fire	46.05 mW
Retire	45.96 mW

Power Results

Conclusion

- The processor successfully accomplishes OoO (out of order) execution, superscalar (2 issue), and pipelining
- The architecture accomplishes this by utilizing a reservation station, ROB (Re-order buffer), free pool, register renaming, and physical registers
- All modules and interfaces were implemented using SystemVerilog and verified using ModelSim in Quartus

References

- This project was primarily created with the knowledge given in Professor Nader Sehatbakhsh's UCLA ECE M116C (Computer Architecture) course.
- Here are links to the lecture videos from Fall 2023 which are most relevant to this project:
  - **[1]** Nader Sehatbakhsh, Superscalar and Intro to OoO, https://www.youtube.com/watch?v= nrJxRNfmgbQ&ab_channel=NaderSehatbakhsh
  - **[2]** Nader Sehatbakhsh, Out of Order Execution, https://www.youtube.com/watch?v=db6UHxfw0C4& ab_channel=NaderSehatbakhsh
  - **[3]** Nader Sehatbakhsh, Cache Design, https://www.youtube.com/watch?v=RwdiDAaK6U8&ab_ channel=NaderSehatbakhsh