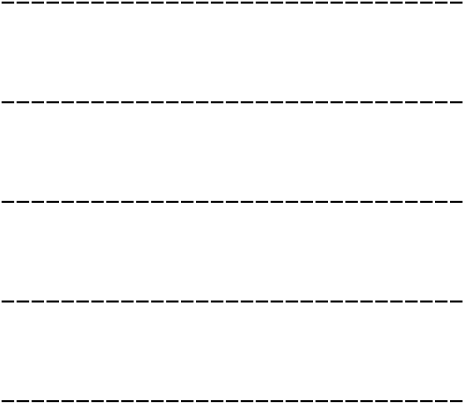
EE 5320 Spring 2017

Advanced Computer Architecture and Arithmetic

Final Project: Simplified CPU

Team Members: Vittal Siddaiah



# Abstract

Reduced Instruction Set Simulation (RISC) microprocessor instruction format enables a conducive design for pipelined implementation. Now-a-days Complex Instruction Set Simulation (CISC) processors are internally implemented as RISC machines, permitting to achieve design speeds in the order of GHz. As part of the computer architecture course work, we have here implemented a four staged simple pipelined architecture using Verilog hardware description language (HDL). The designed four stages includes Instruction Fetch, Decode, Execution and Write Back (F-D-Ex-WB). Design calls for further enhancements to address the data and control hazards.

# Introduction

The design of a CPU is fundamentally affected both by how it will be used and by the characteristics of the underlying implementation technology. Changes in usage or in implementation technology affect the computer design in different ways, from motivating changes in the instruction set to shifting the payoff from important techniques such as pipelining or caching [].

RISC architecture follows a disciplined approach to implementation with fixed instruction set size. This fixed instruction size and format enables easier hardware implementation to enable pipelining, along with simplified compiler designs. RISC architectures also have an increased set of registers as compared to CISC architectures minimizing false dependencies due to limited availability of registers. The RISC-based machines focused the attention of designers on two critical performance techniques, the exploitation of *instruction level parallelism* (initially through pipelining and later through multiple instruction issue) and the use of caches (initially in simple forms and later using more sophisticated organizations and optimizations) [].

# Design Specifications

We set out to design a 32-bit RISC processor with the following design specifications in mind:

* Use of Verilog HDL language to implement a simple 32-bit RISC CPU
* Basic operand format for arithmetic and logic with typical three operand form (two source, one destination)
* Primary memory access is implemented as load/store involving a single register and memory
* A stack will be implemented
* The general-purpose register (GPRs) set should be made as orthogonal as possible with special-purpose registers (e.g. program counter, stack pointer, condition flags, etc.) accessed separately
* The register-to-ALU data path will be a three bus (two source, one destination) structure.

## Register and Data structures

The general-purpose register set is implemented as 32 registers of 32-bits each. The basic data “word” is 32 bits in length with 16-bit half words and 8-bit bytes supported. For all arithmetic/logic instructions any register within this set can be selectable as either source or destination for an instruction. A 32-bit program counter (PC) will be implemented as a special-purpose register. Similarly, a stack pointer (SP), memory address (MA), and memory data (MD) register should be implemented. Access to these special-purpose registers will be limited and they are not available to arithmetic and logic instructions directly. A 32-bit condition flags (CF) register is included as well to represents the conditions – Zero (ZF), Overflow (OF), Carry (CF)

## Memory Subsystem

The memory system is a 32-bit address specifying the byte in memory. Data will be aligned by data size. This will be implemented as a 32-bit wide data path with an effective 30-bit address if the capability to specify aligned half-words and bytes for load and store is implemented. In instructions, addresses are formed either as the contents of a single general-purpose register or the sum of one GP register and a signed immediate value. A small subset of this memory space will be implemented in Verilog to contain a small program (<100 instructions) and a small data set (<100 words)

## Instruction Set

Instruction set is implemented to contain the following types:

* Load register from memory (this is 32-bit data)
* Load register from unsigned memory halfword (zero-extend)
* Load register from unsigned memory byte (zero-extend)
* Load register from signed memory halfword (sign-extend)
* Load register from signed memory byte (sign-extend)
* Load register lower half from immediate halfword
* Load register upper half from immediate halfword
* Load general-purpose register from special-purpose register
* Store register to memory (this is 32-bit data)
* Store register to memory halfword (this is 16-bit data)
* Store register to memory byte (this is 8-bit data)
* Store general-purpose register to special-purpose register
* Push copy of register onto stack
* Pull value on stack into register
* Add unsigned and add signed
* Subtract unsigned and subtract signed
* Negate
* Multiply unsigned, return low 32 bits to destination register
* Multiply unsigned, return high 32 bits to destination register
* Multiply signed, return low 32 bits to destination register
* Multiply signed, return high 32 bits to destination register
* Divide unsigned, return quotient
* Divide unsigned, return remainder
* Divide signed, return quotient
* Divide signed, return remainder
* AND, OR, NOT, XOR
* Compare (operation is subtract and set condition flags but discard difference)
* Bxx *offset* (where xx is condition and signed offset is added to PC if branch is taken)
  + BEQ, BNE, BLT, BLE, BGT, BGE for both unsigned and signed
  + BCC, BCS, BZ, BNZ, BPOS, MNEG, BVC, BVS (for individual condition flags)
  + B (unconditional branch)
* Bsr offset (branch to subroutine, unconditional, saves return address on stack)
* Rts (return from subroutine, reads return address from stack into PC)
* Jump address (replace PC with address)
* JSR address (jump to subroutine, unconditional, saves return address on stack)
* Left shift, left rotate, logical right shift, arithmetic right shift

# Architecture Design and Implementation

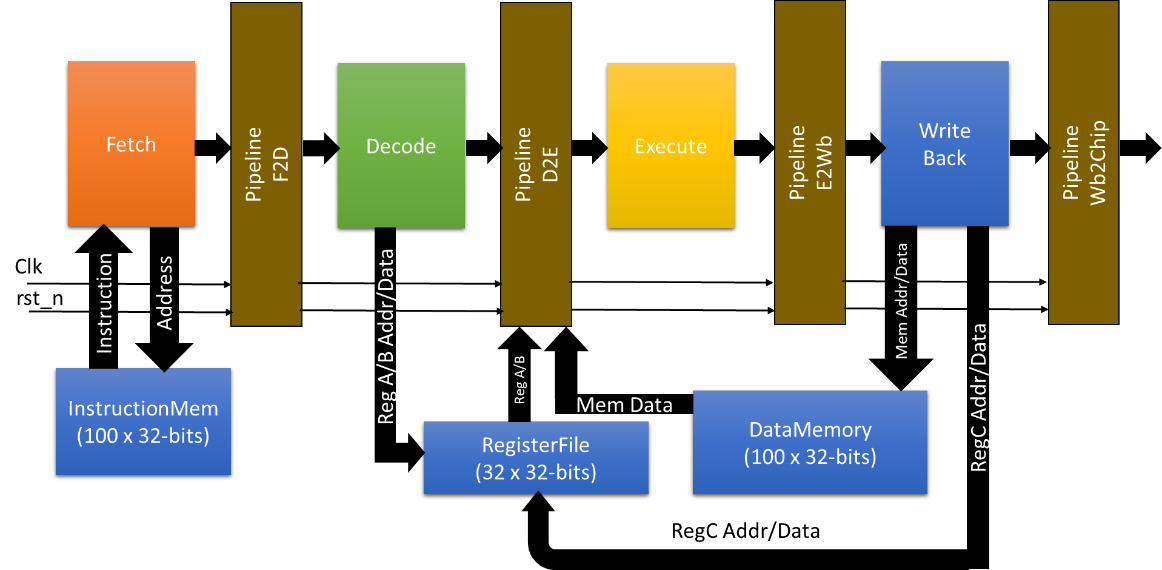
The CPU architecture is designed with pipelined stages approach. The design contains four stages of the pipeline, which are, instruction Fetch, Decode, Execute and WriteBack. Each processing stage is implemented as combinatorial logic with the pipelined registers added between the stages for registered data handshake. Clock and negative edge triggered reset synchronizes the data transitions between the stages. The data moved between the stages includes the data, either immediate, register or memory data, depending on the stage along with the associate instruction. Combining the data with its corresponding instruction helps the design to ensure error-free handover to the next stage.

Figure 1: 32-bit RISC CPU Block Diagram

## Instruction Fetch

Instruction fetch engine is designed to handle the instruction memory addressing and forwarding the read instruction to the decode engine. Instruction memory is implemented as 100x32-bits sized byte addressable memory. To ensure the instruction alignment the lower two bits of the address are always appended to zeros forcing a 32-bit word alignment. As the implementation is based of RISC architecture a fixed 32-bit sized instruction set usage ensures the memory alignment to 32-bit word size.

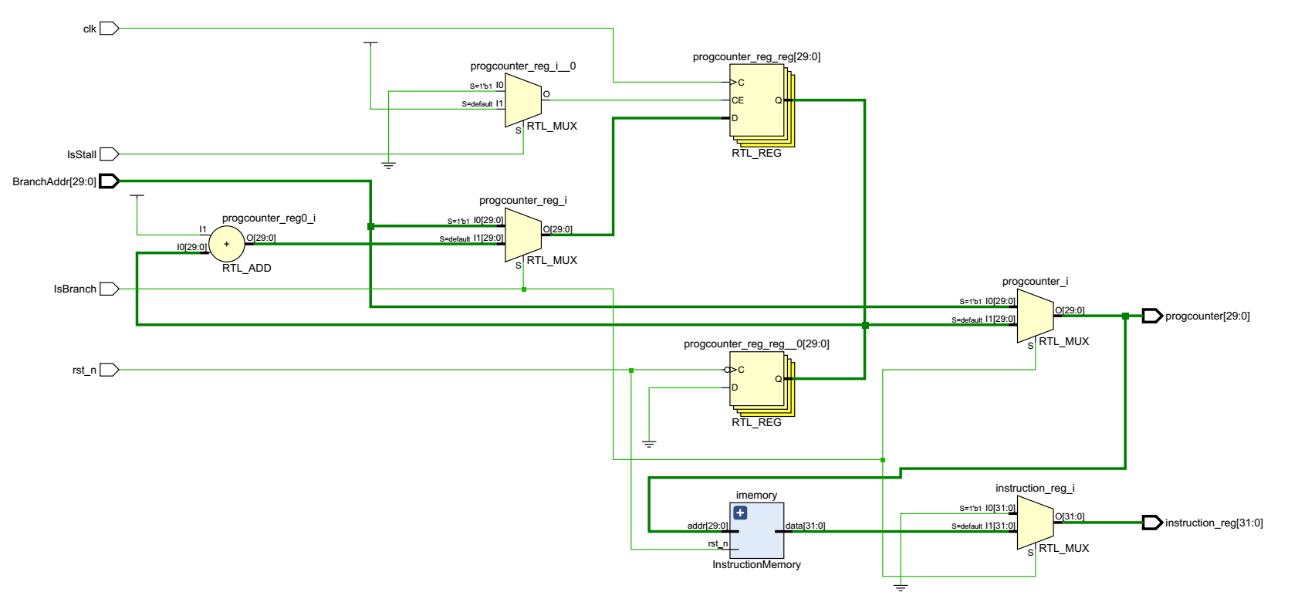
The instruction address is incremented every clock to point to next instruction. However, a mux handles updating the instruction address during a branch or jump instruction, based on the control signal which indicating the occurrence of such an event.

Figure 2: Instruction Fetch: Implementation Schematic

## Instruction Decode

The system data path is designed to contain 3 data buses dataA, dataB, and dataC. The A and B buses are the source data buses which carry the data across the processing stages and the C bus carries the destination data to be updated either in memory or register.

Instruction decode engine takes in the instruction and produces data source addresses. If the input is register based instruction, the decode engine produces two register source addresses and if it involves memory data, corresponding address is generated. The pipeline stage from decode to execute, gets updated with the required data from either the memory or register file.



Figure 3: Instruction Decode: Architectural Diagram

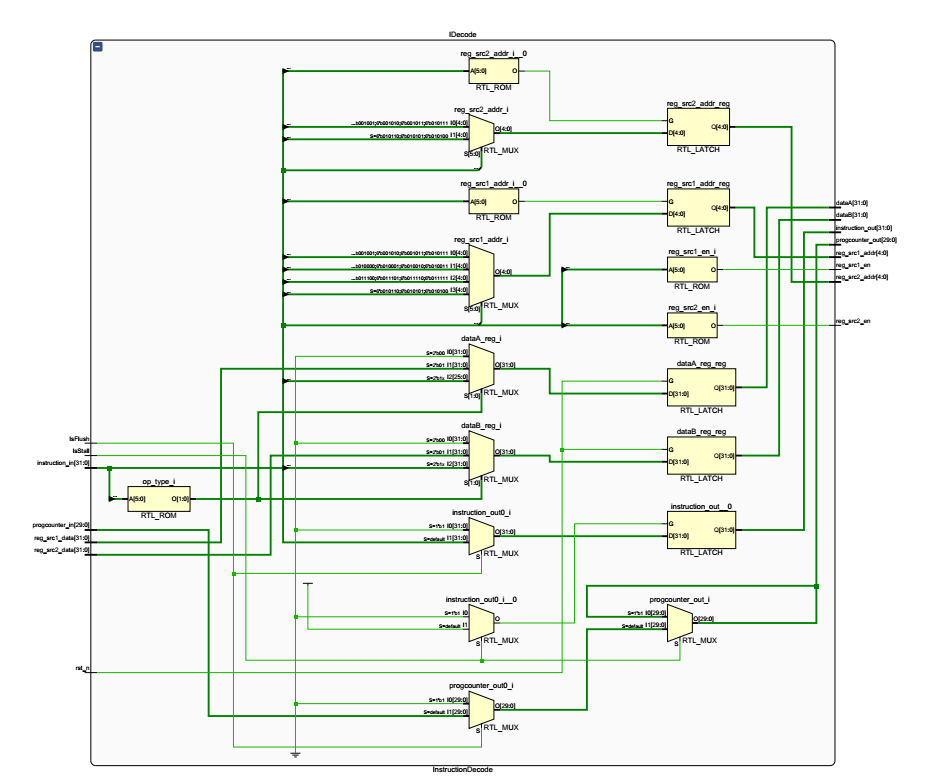


Figure 4: Instruction Decode: Implementation Schematic

## Instruction Execution

Tools

Test Bench

Simulation Results

Synthesis Results

Summary