Computer Architecture

Verilog Project: Single Cycle MIPS Processor

Dr. Khunjush

Spring 2025

Contents

Introducing the project: Single Cycle MIPS

By now, you are familiar with the concept of single cycle processors. These processors do every instruction in one clock cycle. The Goal of this project is to implement a 32-bit single cycle processor using Verilog hardware description language.

1 Part I: The Basics

In this first phase, you will design and implement a 32-bit single-cycle MIPS-like processor in Verilog. You must support the following instruction categories:

1. **R-Type Arithmetic/Logical**:
   - `add`, `sub`, `mul`, `and`, `or`, `xor`, `shift`, `rotate`
2. **I-Type Arithmetic/Memory**:
   - `addi`, `subi`, `lw`, `sw`
3. **Control Transfer**:
   - `jmp`, `beq`
4. **Custom Crypto** (see Part II):
   - `enc`, `dec`

1.1 Instruction Format & Bit Partitioning

Your instructions must be exactly 32 bits wide. You have flexibility in choosing how many bits to allocate for:

- `opcode` (specifies the instruction),
- `rs`, `rt`, `rd` (register specifiers),
- `immediate` values (for I-type),
- and (for R-type) a `funct` field if desired.

You should justify your choices for how many bits you associate with each field.

1.2 Datapath & Control Signals

Your CPU should comprise at minimum these modules (with at least one top-level wrapper that instantiates them all):

- **Instruction Memory (IMEM)**: Holds your assembled machine code.
- **Data Memory (DMEM)**: Supports `lw` and `sw` (32-bit word).
- **Register File (RF)**: 32 registers, each 32 bits wide. (Incldуing a zero register)
- **Program Counter (PC)**: 32-bit register; updates on each clock's rising edge.
- **Arithmetic Logic Unit (ALU)**: Performs 32-bit arithmetic/logic; should support `add`, `sub`, `mul`, `and`, `or`, `xor`, `shift`, `rotate`.
- **Sign-Extend Unit**: Extends 16-bit immediates to 32 bits (for `addi`, `subi`, `lw`, `sw`, `beq`).
- **Control Unit (CU)**: Decodes `opcode` (and `funct` for R-type) to generate control signals:

$$\texttt{RegDst},\ \texttt{ALUSrc},\ \texttt{MemtoReg},\ \texttt{RegWrite},\ \texttt{MemRead},\ \texttt{MemWrite},\ \texttt{Branch},\ \texttt{Jump},\ \texttt{ALUOp}[1:0]$$

- **Helper Modules**: Multiplexers (e.g. a 2:1 MUX for selecting ALU input between `rt` and immediate), shift/rotate-by-amount module, etc.
- **Top-Level Datapath**: A Verilog file (e.g. `SingleCycleMIPS.v`) connecting IMEM, DMEM, RF, ALU, PC, Sign-Extend, CU, and all multiplexers.
- **Testbenches**: For each module and for the full processor—include at least the following scenarios:
  1. ALU: add, sub, mul, boundary values (e.g. 0x7FFFFFFF + 1), shift/rotate edge cases.
  2. RF: read/write same register, simultaneous reads, zero register behavior.
  3. Memory: lw/sw to different addresses (aligned vs. misaligned), read after write.
  4. CU: each opcode/funct combination, illegal opcodes should default to "NOP" or raise an error.

2 Part II: Encrypt and Decrypt

In this phase, you will extend your ISA with two new single-cycle R-type instructions—`enc` and `dec`—that perform a symmetric transformation on a 32-bit word using a 32-bit key. Both instructions use the same key: applying `dec` to a word that was previously processed by `enc` (with that key) will recover the original word.

Choose two unused opcodes to distinguish `enc` versus `dec`.

The registers have the following roles:

- `rs`: Contains the input word (plaintext for `enc`, ciphertext for `dec`).
- `rt`: Contains the 32-bit key.
- `rd`: Destination register, which will hold the transformed output.

Because the transformation is symmetric, the datapath for `enc` and `dec` is identical. The Control Unit will assert a new control signal whenever `opcode = enc` or `opcode = dec`, and the ALU (or a dedicated combinational block) will apply the same reversible operation in one cycle.

3 Part III: Inner Product

In this phase, you will write a machine-code program (to be stored in your Instruction Memory) that computes the inner product (dot product) of two 4-element integer vectors. The inner product of two length-4 vectors $\mathbf{A} = [a_0, a_1, a_2, a_3]$ and $\mathbf{B} = [b_0, b_1, b_2, b_3]$ is defined as:

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=0}^{3} a_i \times b_i = a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3.$$

In other words, you multiply each corresponding pair of elements and accumulate the results into a single 32-bit sum.

3.1 Fixed Vector Length and Data Layout

For Part III, both vectors have fixed length $n = 4$. In your Verilog testbench (e.g. `tb_InnerProduct.v`), initialize Data Memory so that it contains two 4-element vectors of 32-bit words. For example:

- Store vector A at consecutive addresses starting at base address BASE_A. That is:

  $\texttt{MEM[BASE\_A+0]} = a_0, \quad \texttt{MEM[BASE\_A+4]} = a_1, \quad \texttt{MEM[BASE\_A+8]} = a_2, \quad \texttt{MEM[BASE\_A+12]} = a_3.$

- Store vector B at consecutive addresses starting at base address BASE_B. That is:

  $\texttt{MEM[BASE\_B+0]} = b_0, \quad \texttt{MEM[BASE\_B+4]} = b_1, \quad \texttt{MEM[BASE\_B+8]} = b_2, \quad \texttt{MEM[BASE\_B+12]} = b_3.$

Choose two distinct 32-bit addresses in your data memory for BASE_A and BASE_B (for example, BASE_A = 0x00000100 and BASE_B = 0x00000110), and document these in your testbench comments.

3.2 Registers and Loop Structure in Machine Code

You will write machine code instructions directly into Instruction Memory (for example, in an initial block that uses IMEM[0] = 32'b...;). Your program should use registers similarly to the outline below:

- $s0: Holds BASE_A (base address of vector A).
- $s1: Holds BASE_B (base address of vector B).
- $s2: Holds the constant length $n = 4$.
- $s3: Loop counter / index (initially 0, up to 3).
- $s4: Holds the current element A[$s3] loaded from memory.
- $s5: Holds the current element B[$s3] loaded from memory.
- $s6: Accumulator register (initialize to 0).
- $t0, $t1, ...: Any additional temporaries you need (document usage).

Because you are writing machine code, you must assemble each instruction into its 32-bit binary encoding.

4 Bonuses

The following parts are not mandatory to the project, but implementing them will result in bonus score.

4.1 ALU Control

Implementing ALU Control will have bonus points. The design, number of required bits, and all other parameters are up to you.

4.2 automated dot product

For Part III, you can write a program (in any programming language) that can automate the initialization part for both your data and instruction memory. Doing this step might even be easier than writing the code from scratch in verilog.

5 Final Remarks

You should submit your **complete code** as well as **complete documentation** as a single zip file. We will announce the presentation time of your projects at a later date. You can use any software to write your Verilog modules, but using Xilinx Vivado Design Suite is preferred. For the final presentation, please be ready to run your module on your own machines.

- Upload your project here before the deadline.
- File format name should be : `NameLastName_StudentID_CA_MipsProject.zip`.
- Your uploaded zip file should contain all your codes the report file.
- Please ensure that your submission reflects your own analysis; copying another student's project constitutes a breach of academic integrity and undermines the learning objectives of the project.
- Feel free to ask any questions.

Keep on keeping on

Ahmad Shams, Sina Hakimzade, Melika Ganji, Niloofar Naghibi,
Raha Rahmanian, Melika Zamani, Arman Alavi