CSE 490/590 Computer Architecture, Spring 2025

Project 1 - Design of a 16-bit Processor (non-pipelined)

ASSIGNMENT	DUE DATEs
Group Details	Friday February 28, 2025
Code Checkpoint	Friday March 14, 2025
Working Code with Verilog Simulation	Wednesday March 28, 2025
Project Demos	April 4 – April 7, 2025
Working Code on Xilinx Hardware and Final Report	Monday April 7, 2025

1 – Synopsis

In this group project your team will implement, simulate, and synthesize a simple 16-bit processor.

To implement the processor you will use Xilinx Vivado and Verilog:

- Verilog is a Hardware Description Language (HDL), which is used to specify the structure and behavior of a hardware design.
- Vivado is an integrated development environment from Xilinx which includes many tools for the development, analysis, and synthesis to target hardware.

The Verilog code you write will implement the features of the 16-bit processor described in this document. Key tasks include:

1. Synthesize your code and run a software simulation of your design to verify its functionality.
2. Synthesize your code to create what is known as a "bitstream" to target a Xilinx FPGA board, and program the FPGA board with that bitstream to test and verify the functionality on real hardware.

2 – Tools and Resources

For information about how to install and get started with Vivado, see the appendices listed below (on the class website):

Appendices

- **Appendix 1:** includes instructions on how to install and use Vivado locally and remotely
- **Appendix 2:** includes instructions on how to create a project in Vivado
- **Appendix 3:** includes instructions on how to program the Basys 3 FPGA board.
    - Available on the course website

For information about the Xilinx Basys 3 FPGA Board:
https://digilent.com/reference/_media/reference/programmable-logic/basys-3/basys3_rm.pdf

3 – Group Details

This is a group project. Each group should have four students, but having three may be allowed with permission from course staff.

Please form groups as soon as possible!

Submit your group details on UBLearns!

GROUP DETAILS ARE DUE: February 28, 2025.

4 – Deliverables

This project has four (4) deliverables:

- Verilog Source Code for Working Simulation (The Verilog code that implements your 16-bit processor)
- If needed, updated working code used for hardware verification and demonstration.
- A Final Report (Diagrams, descriptions, and simulation results of your processor implementation)
- Live Interview and Hardware Demonstration; this will apply to each team member on an individual basis. We will ask you questions about your work on the project and you will demonstrate the functionality of your processor in simulation and on the FPGA.

4.1 – Code Checkpoint

Upload a short report, **as a PDF**, documenting the work you have completed so far (e.g. descriptions of what components have been worked on). There must also be a brief section at the beginning outlining the individual contributions of each teammate so far. This code checkpoint will **not** be harshly graded, we are simply looking for evidence of at least some progress from your team.

4.2 – Verilog Source Code for Working Simulation

You will use Verilog to describe the structure and behavior of your 16-bit processor. The individual components of the processor (see the list of required components in the processor specifications section) should be designed using **behavioral Verilog** (you may also use structural Verilog as well, but it is not recommended)**.** The processor as a whole (how all the individual components are connected together) **must** be designed with **structural Verilog.**

4.3 – Updated Verilog Code for Working Hardware Verification

Once you have your design and simulation completed, you will then generate a bitstream and download it to an FPGA board. As needed, make changes to your code to generate the bitstream and to verify the functionality of your cpu implementation on hardware.

4.4 – Final Report

Your final report will be a formal document which demonstrates the work you have done and explains the implementation details of your processor.

The final report MUST include the following:

- Schematic of your datapath and control path
- Short description of every component in your design
- Simulation results for each component
- Work distribution (how much work each team member contributed towards the project)
- References (if applicable)

4.5 – Live Interview and Hardware Demonstration At the end of project 1, you and your teammates will have a live "interview" and demonstration. Each team member will be interviewed and asked questions about the processor your team implemented. Your team will also have to demonstrate the functionality of your processor and prove that it is correctly implemented. When demonstrating the functionality of your processor, you will need to show both the simulated results of running your processor and a hardware demonstration where you will run your processor directly on an FPGA board.

5 – Processor Specifications

5.1 – Overview

The 16-bit processor is of a simple, single-cycle, non-pipelined design.

- Word Size: 16 bits
- Memory is addressed by: byte (8 bits)
- Non-pipelined

5.2 – Required Components

Your processor will have multiple individual components that will be combined using structural Verilog. There are specific components that you must implement in your design: **Required Components:**

- Instruction Memory
- Data Memory
- Register File
- ALU
- Sign Extension
- Control Unit(s)
- Multiplexer(s)
- Program Counter

You may implement additional components as needed by your design.

5.3 – Component Information

- The Register file has sixteen registers, $s0 through $s15 all of which are 16 bits (2 bytes) in size.
- The ALU does not need to handle carry and overflow conditions.

- The Instruction Memory has a minimum size of 64 memory locations.
- The Data Memory has a minimum size of 64 memory locations.

5.4 – Instructions

5.4.1 – Instruction Formats

R-type instruction:

opcode				**rt/rd**				**rs**				**function code**			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

I-type instruction:

opcode				**rt/rd**				**rs**				**immediate**			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

J-type instruction:

opcode				**address**											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

5.4.2 – Mandatory Instructions to Implement

Instruction	**Opcode (in binary)**	**Function Code (in binary)**	**Type**
add	0000	0000	R-type
sub	0000	0001	R-type
sll	0000	0010	R-type
and	0000	0011	R-type
lw	0001	N/A	I-type
sw	0010	N/A	I-type
addi	0011	N/A	I-type
beq	0100	N/A	I-type
bne	0101	N/A	I-type
jmp	0110	N/A	J-type

Note: The 16-bit instruction format allows for many more instructions. Students are welcome to add any additional functionality to their processor, but the above instructions must be implemented as listed. Only the 10 instructions listed here will be used when evaluating the processor.

5.4.3 – Detailed Instruction Information

add
The add instruction interprets the values of both the registers R[rt/rd] and R[rs] as signed 16-bit integers, computes R[rs]+R[rt/rd] (their sum) and stores the result as a signed 16-bit integer in the register R[rt/rd]. Overflow is undefined behavior (don't worry about it).

sub
The sub instruction interprets the values of both R[rt/rd] and R[rs] as signed 16-bit integers, computes R[rs]-R[rt/rd] (their difference), and stores the result as a signed 16-bit integer in the register R[rt/rd]. Underflow is undefined behavior (don't worry about it).

sll
The sll instruction interprets the values of both R[rt/rd] and R[rs] as unsigned 16-bit integers and shifts the bits in R[rt/rd] by R[rs] bits with zeros inserted on right to replace the shifted bits. The leftmost shifted bits are discarded. The result is stored in the register R[rt/rd]. Example: If R[rt/rd]=0b00000000 01011010 and R[rs]=4, then executing sll would result in: R[rt/rd]=0b00000000 10100000

and
The and instruction takes the values of R[rt/rd] and R[rs], performs bitwise-and, and stores the result as an unsigned 16-bit integer in the register R[rt/rd]. Example: If R[rt/rd]=0b00000000 00001010 and R[rs]=0b00000000 00001100, then executing and would result in: R[rt/rd]=0b00000000 00001000.

addi
The addi instruction interprets the value of R[rs] as a 16-bit signed integer and the value of immediate as a 4-bit signed integer, computes their sum, and stores the sum as a signed 16-bit integer in the register R[rt/rd].

sw
The sw instruction retrieves the value of R[rs] as an unsigned 16-bit integer and the value immediate as a signed 4-bit integer, then stores the value in R[rt/rd] at the memory address R[rs]+immediate. The value is stored in memory using big-endian.

lw
The lw instruction retrieves the value of R[rs] as an unsigned 16-bit integer and the value immediate as a signed 4-bit integer and then loads a word from memory at the address R[rs]+immediate and stores it in the register R[rt/rd]. The value is stored in memory using big-endian.

beq
The beq instruction interprets the values of both R[rt/rd] and R[rs] as signed 16-bit integers and checks if the two values are equivalent. If they are equivalent, the immediate value is interpreted as a signed 4-bit integer. The immediate value is shifted left 1 bit and added to the program counter, where the program counter is the address of the next instruction to be executed. Example: If R[rt/rd]=0b00000000 00001010 and R[rs]=0b00000000 00001010 with the program counter at instruction address 0b00000000 00001110 (the currently executing instructions address + 2) and immediate=0b0010, then executing would result in the program counter branching to instruction address 0b00000000 00010010.

bne
The bne instruction interprets the values of both R[rt/rd] and R[rs] as signed 16-bit integers and checks if the two values are equivalent. If they are not equivalent, the immediate value is interpreted as a signed 4-bit integer. The immediate value is shifted left 1 bit and added to the program counter, where the program counter is the address of the next instruction to be executed. Example: If R[rt/rd]=0b00000000 00000010 and R[rs]=0b00000000 00001010 with the program counter at instruction address 0b00000000 00001110 (the currently executing instructions address + 2) and immediate=0b0010, then executing would result in the program counter branching to instruction address 0b00000000 00010010.

jmp
The jmp instruction interprets the address value as a 12-bit signed integer. The integer is shifted left 1 bit then added with the program counter and the code continues executing from that address. Example: If the program counter is at instruction address 0b00000000 00001110 (the currently executing instructions address + 2) and the address=0b0000 00000010, then executing would result in the program counter branching to instruction address 0b00000000 00010010

6 – Frequently Asked Questions

6.1 – Where do I start?

Start by first designing your datapath for your CPU. Then, start designing major components such as the program counter, ALU, instruction memory, etc. Smaller components such as Sign Extension and Muxes can be implemented as needed. Also start with R-type instructions first, and then later work on implementing I-type instructions. Once the branch I-type instructions are implemented, the J-type instruction can be implemented.

6.2 – How do I design memory (Instruction/Data)?

6.2.1 – Initializing Memory in Verilog

Option 1:

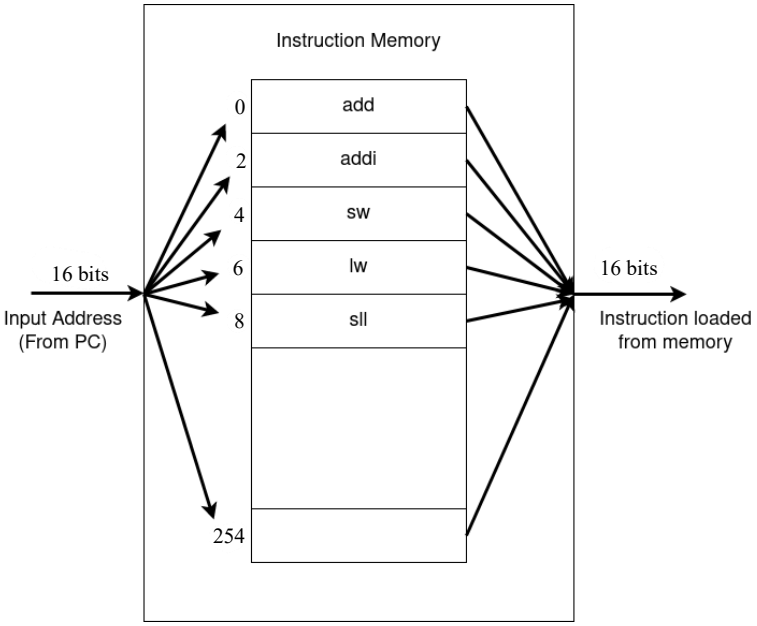
When initializing memory in Verilog, you may use the `readmemb` and `readmemh` commands, both of which read a text file and uses the contents of the text file to initialize an array/memory.

Option 2:

You can simply hardcode initial values for memory by assigning values to memory inside of the initial begin block.

6.2.2 – Instruction Memory Design

In instruction memory, you get an address/index as input and look up the for the opcode of the instruction stored in that address/index of the memory/array (like the figure below):



6.2.3 – Data Memory Design

When designing the Data Memory:

1. Initialize an array with appropriate values.
2. Be sure to consider read and write operations:
    - Read -> Take an address/index of a memory/array as input and retrieve the data in that location and send it as output
    - Write -> Get the data and address/index as input and store it in an array/memory.
3. The control lines should choose between Read and Write depending on the instruction.

6.2.4 – Memory Representation in Verilog

For behavioral Verilog representation, you can consider memory as an array.

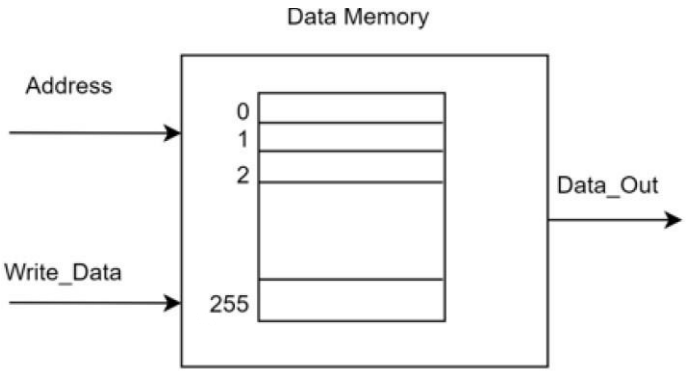


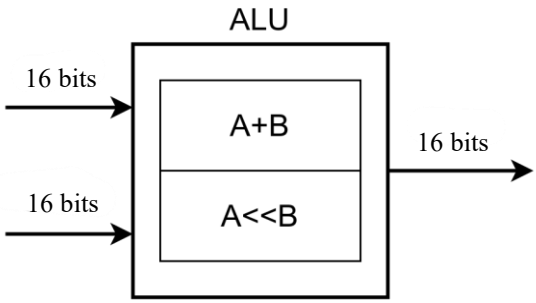
Fig 2. Data Memory Representation

6.2.5 – Why are there minimum and maximum memory sizes?

The 64 byte minimum was arbitrarily chosen. The 65536 byte maximum is a limitation inherent of the processor design itself. Since the processor in this project is a 16-bit processor, memory addresses can only be encoded with 16 bits, allowing for $2^{16} = 65536$ unique addresses. Since the processor is byte-addressed, this means that memory can be *at most* 65536 bytes large. The Basys3 Board has limited memory on it. Including 65536 memory locations for both the Instruction Memory and Data Memory may be too much. We recommend keeping the actual number of locations as low as possible, such as 256 bytes. We will only test up to 64 byte memory size.

6.3 – What type of Adder? Carry or Overflow?

A simple adder as shown below is sufficient. Carry and overflow conditions do NOT need to be considered. (The design below is an example and may not exactly reflect what you need to implement).



6.4 – How exactly do you code/implement an instruction?

The opcode of each instruction is given in the project description. You may start with one type of instruction. If you start with R type (for example, add), store the opcode of an instruction in a memory location/array in your instruction memory. Then design the register file with just support for this instruction, followed by the ALU unit. Then design the control unit such that it can send control signals to these individual components just for the add instruction. Once add is implemented and verified to work correctly, you can update the components you designed to

support another instruction (such as addi). Also add additional components, such as mux(es) as needed. (Other instructions may need additional components such as sign extension for I type instructions.)

6.5 – Why do we shift the immediate value of the branch/jump instructions?

Like in MIPS, memory for this processor is byte addressable. Since each instruction is 2 bytes, any branch or jump instructions will only jump an even number of bytes. By shifting the immediate value left 1, we can effectively double the range of the jump since we know the LSB will always need to be a 0. This also keeps the program counter aligned.