Laboratory 5: Simple CPU and Memory Mapped I/O

Objectives

1. Able to implement Simple CPU in VerilogHDL
2. Able to implement the memory mapped I/O

Background

Memory Mapped I/O and Port-Mapped I/O

Memory Mapped I/O and Port-Mapped I/O are two different methods for implementing input/output between CPU and peripheral devices in a computer.

Memory Mapped I/O uses the same address to address both memory and I/O devices. The memory and registers of devices are associated with physical addresses. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register. This type of I/O allows software to interact with I/O directly using standard load and store instructions.

Unlike the Memory Mapped I/O, Port-mapped I/O uses special CPU instructions for performing I/O, such as the inp and outp. The benefit of Port-mapped I/O is the separation of address spaces between I/O and memory.

To connect a device to either memory mapped I/O or port-mapped I/O, a decoder must be added to the address. When an address is matched, accessing (reading and writing) to the data bus will be relayed to a device. In certain cases (eg. microcontroller), control registers (and buffers) will be added to related addresses in order to act as an interface between processor and devices.

(For more information, see the video clips.)

Exercises

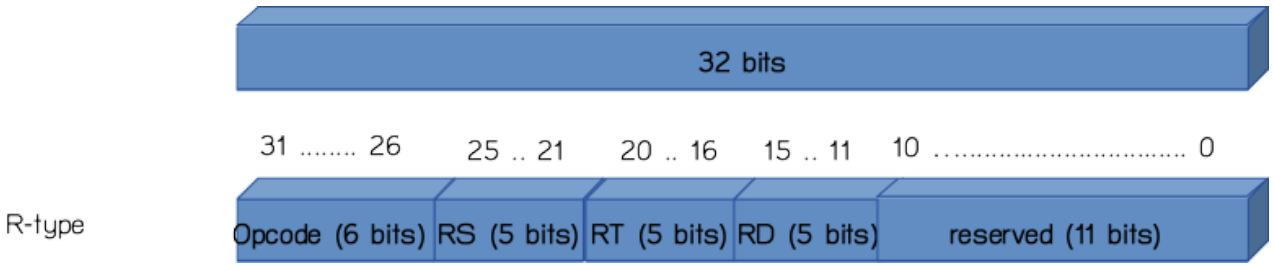
Please use nanoLADA CPU code[6] as a base.

1. Based on the given ALU, extend the ALU to support the following operations.

ALU OP	Operations
000	{Cout,S}=A+B+Cin;
001	{Cout,S}=A-B;
010	S=A | B; Cout=0; (or)
011	S=A & B; Cout=0; (and)
100	S=A ^ B; Cout=0; (xor)
101	S=-A; Cout=0; (2's complement)
110	S=~A; Cout=0; (not)
111	S=~B; Cout=0; (not)

Use the simulator to validate your ALU.

2. . Extend the R-type instruction format to support the following operations.



[6] Available at https://www.cp.eng.chula.ac.th/~krerk/books/Computer%20Architecture/nanoLADA/

Original instruction

	Opcode (6 bits)	RS (5 bits)	RT (5 bits)	RD (5 bits)	Reserved (11bits)
ADD rd, rs, rt R[rd] ← R[rs] + R[rt];	000001				

New instructions.

	Opcode (6 bits)	RS (5 bits)	RT (5 bits)	RD (5 bits)	ALU Operation (11 bits)
ADD rd, rs, rt R[rd] ← R[rs] + R[rt];	000001				00000000_000
SUB rd, rs, rt R[rd] ← R[rs] - R[rt];	000001				00000000_001
OR rd, rs, rt R[rd] ← R[rs] | R[rt];	000001				00000000_010
AND rd, rs, rt R[rd] ← R[rs] & R[rt];	000001				00000000_011
XOR rd, rs, rt R[rd] ← R[rs] ^ R[rt];	000001				00000000_100
COM rd, rs,xx R[rd] ← -R[rt];	000001				00000000_101
NOT rd, rs, xx R[rd] ← ~R[rs]	000001				00000000_110

Use the simulator to validate your extended CPU.

3. Use the knowledge from memory-mapped I/O to map the following devices to associated addresses. (You have to add a few registers and buffers to the associated address.) Note that you have to take away memory from those addresses.

Address	Device	Note
0xFFF0	4-bit register for driving Seven-segment display digit 0.	Use last 4 bits
0xFFF4	4-bit register for driving Seven-segment display digit 1.	Use last 4 bits
0xFFF8	4-bit register for driving Seven-segment display digit 2.	Use last 4 bits
0xFFFC	4-bit register for driving Seven-segment display digit 3.	Use last 4 bits
0xFFE0	4-bit buffer for reading Switch [3..0] (A)	Use last 4 bits
0xFFE4	4-bit buffer for reading Switch [7..4] (B)	Use last 4 bits
0xFFE8	4-bit buffer for reading Switch [11..8] (Op)	Use last 4 bits

Write a simple software to your ROM to repeatedly read from switches. Display the value of switch [3..0] to seven-segment display digit 0. Display the value of switch [7..4] to seven-segment display digit 1. For seven segment display digits [3..2], show the result from the following operations.

Switches [10..8] (Op)	**Seven segment display [3..2]**
000	Show A+B
001	Show A-B
010	Show A | B
011	Show A&B
100	Show A^B
101	Show ~A (not A)
110	Show -A
111	Show -B

Demonstrate your design in our BASYS 3 FPGA board.