CS220 SEM II 2022-23

CSE- Bubble Processor
Assignment7

Pratham Sahu - 210755
Shantanu Kolte - 210958

Date: April 14, 2023

Contents

Overview

The given code is a Verilog hardware description of a simple Processor. It consists of several modules - VEDA, adds, alu, branch, and decoder - each of which has a specific function in the processor.

The VEDA module is the memory unit that holds the instructions and data to be used by the CPU. It has a 128x32-bit memory block implemented as an array of registers. The module has an input port to receive data from the external world and another port to send data to the external world. The $data_{in}$ and $data_{out}$ ports are used to read from and write to the memory, respectively. The module also has an input port to receive the address of the memory location being accessed, and an input port to receive a write-enable signal.

The alu module is another arithmetic logic unit that performs various arithmetic and logical operations, such as addition, subtraction, AND, OR, shift left, shift right, and comparison. The module has several input ports to receive the operands and control signals needed for each operation, and an output port to send the result of the operation to the rest of the CPU.

The decoder module is responsible for decoding the instruction being executed. It has an input port to receive the instruction code, and output ports to send signals that determine the type of instruction being executed and the control signals needed to execute the instruction.

PDS1: Decide the registers and their usage protocol.

In the first step, we need to decide on the registers and their usage protocol for the CSE-BUBBLE processor. The register file is used to store the operands of the instructions and intermediate results generated by the instructions.

The CSE-BUBBLE processor has a 32-bit register file with 32 registers, each of which can store a 32-bit value.

The register file has the following registers:

$zero: a fixed register whose value is always zero
$at: reserved for the assembler
$v0, $v1: used to store function return values
$a0-$a3: used to pass arguments to functions
$t0-$t9: temporary registers that can be used by the programmer
$s0-$s7: saved registers that must be preserved across function calls
$k0, $k1: reserved for the kernel
$gp: global pointer
$sp: stack pointer
$fp: frame pointer
$ra: return address

The usage protocol for the registers is as follows:

The registers $zero, $at, $k0, and $k1 are read-only and cannot be written to.

The registers $v0 and $v1 are used to store the return values of functions.
The registers $a0-$a3 are used to pass arguments to functions.
The registers $t0-$t9 can be used by the programmer as temporary registers.
The registers $s0-$s7 are saved registers that must be preserved across function calls.
The register $gp is used to store the global pointer. The register $sp is used to store the stack pointer.
The register $fp is used to store the frame pointer. The register $ra is used to store the return address.

PDS2: Decide upon the size for instruction and data memory in VEDA.

The ISA we used is a modified version of the MIPS32 ISA. It has been modified slightly to also include pseudo instructions 'bgt', 'bgte', 'ble' and 'bleq', to make sure they are executed in 1 clock cycle as per the requirements of the assignment

For this step, we need to decide upon the size of the instruction and data memory. The instruction memory will store the instructions that the processor will execute, and the data memory will store the data that the instructions will operate on.

The size of the memory will depend on the size of the instructions and the amount of data that we need to store. We will use 32-bit instructions and 32-bit data values. We will start with a small memory size and increase it as needed during the implementation and testing phases. We initially started with a memory size of height 128 and width 32 bits that was enough to store the bubble sort code at the end.

PDS3: Design the instruction layout for R-, I- and J-type instructions and their respective encoding methodologies.

In this step, we will design the instruction set architecture for CSE-BUBBLE processor. We will be using three different instruction types: R-type, I-type, and J-type.

- **R-Type Instructions**: These are register-to-register instructions where the operation is performed on two register operands and the result is stored in a third register. The following is the instruction layout for R-type instructions:

31-26	25-21	20-16	15-11	10-6	5-0
Opcode	rs	rt	rd	shamt	funct

Table 1: R-type instruction set

  - Opcode (6 bits): Specifies the type of R-type instruction.
  - rs (5 bits): Specifies the first register operand.
  - rt (5 bits): Specifies the second register operand.
  - rd (5 bits): Specifies the register where the result is stored.

- shamt (5 bits): Specifies the amount of shift to be performed (for shift instructions).
- funct (6 bits): Specifies the specific function to be performed by the instruction

- **I-Type Instructions**: These are immediate instructions where the operation is performed on one register operand and one immediate value. The following is the instruction layout for I-type instructions:

31-26	25-21	20-16	15-0
Opcode	rs	rt	Immediate-Value

Table 2: I-type instruction set

  - Opcode (6 bits): Specifies the type of I-type instruction.
  - rs (5 bits): Specifies the register operand.
  - rt (5 bits): Specifies the register where the result is stored.
  - Immediate Value (16 bits): Specifies the immediate value used in the operation.

- **J-Type Instructions**: These are jump instructions where the program jumps to a specific memory location. The following is the instruction layout for J-type instructions:

31-26	25-0
Opcode	Jump Address

Table 3: J-type instruction set

  - Opcode (6 bits): Specifies the type of J-type instruction.
  - Jump Address (26 bits): Specifies the memory address where the program should jump.

Now that we have designed the instruction layout, we need to decide on the encoding methodologies for each instruction type. We will be using the following encoding methodologies:

- R-Type Instructions: The encoding of R-type instructions will be done by concatenating the Opcode, Rs, Rt, Rd, Shift, and Funct fields in that order.
- I-Type Instructions: The encoding of I-type instructions will be done by concatenating the Opcode, Rs, Rt, and Immediate Value fields in that order.
- J-Type Instructions: The encoding of J-type instructions will be done by concatenating the Opcode and Jump Address fields in that order.

We will be using a 32-bit word size for each instruction and data memory location.

PDS4: Now design and implement an instruction fetch phase where the instruction next to be executed will be stored in the instruction register.

To fetch the instructions we use the PC or program counter register. When we finish executing a instruction, we shift the program counter to the next instruction and fetches that from the VEDA memory. This fetched memory is stored in the instruction register(IR).

This is done by assigning the PC as an input to the memory_instr instantiation of the VEDA memory module, and IR an output.

PDS5: Next design and implement a module for instruction decode to identify which data path element to execute given the opcode of the instruction.

The decoder module takes the current instruction in the instruction register (IR) as an input and outputs two signals, type1 and type2. Type1 indicates the type of instruction (R-type, I-type, or J-type), and type2 identifies the specific instruction within that type.

The decoder module uses a combinational logic circuit to analyze the opcode of the instruction and determine its type and specific instruction. It takes the first six bits of the instruction, which represent the opcode, and checks them against predefined values for each type of instruction. Based on the opcode, the module assigns the appropriate type1 and type2 values.

The decoder module is essential for the instruction fetch phase of the processor. It allows the processor to determine the type of instruction in the IR, which is necessary for executing the instruction correctly. With this module, the processor can fetch the next instruction and store it in the IR for decoding and execution.

Overall, the decoder module is a crucial part of the processor, and its successful implementation ensures that the processor can perform the instruction fetch phase accurately.

The decoded information is then passed on to the rest of the processor to carry out the appropriate actions.

To design this module, we need to look at the instruction set architecture (ISA) of the processor and determine how the opcode and operands are encoded which we have already decided in PDS3. By looking at specific bits of the IR, we can identify which registers are required and hence use them as needed.

PDS6: ALU Design

The ALU module is designed for the various arithmetic and logical instructions which our PC will need to be executed. The following instructions described in the ISA in the original question will be executed in the ALU :-

- *add r0, r1, r2*
- *sub r0, r1, r2*
- *addu r0, r1, r2*
- *subu r0,r1,r2*
- *addi r0,r1,1000*
- *addiu r0,r1, 1000*
- *and r0,r1,r2*
- *or r0,r1,r2*
- *andi r0,r1, 1000*
- *ori r0,r1, 1000*
- *sll r0, r1, 10*
- *srl r0, r1, 10*
- *slt r0,r1,r2*
- *slti 1,2,100*

The ALU has 7 inputs and 1 output, which are broken down as follows:-

alu_en –> Turns the ALU on and off as per need. If alu_en=0, whatever the inputs be, the output will not change

type1, type2 –> Decoded instructions from the CPU, they will be used by the ALU to identify what operation needs to be carried out on RS and RT

RS, RT –> The 2 operands

SHAMT –> Used during sll and srl instructions (As per our ISA)

IMM –> Used for I-type instructions

RD–> Is the lone output which is the result of the Arithmetic or Logical Operation

PDS7-8: Implementing the FSM in the CPU module along with the branching instructions

The main glue of the CSE-Bubble code is the FSM which controls all the different signals used for carrying out different instructions like updating the Program Counter, and carrying out the *lw* and *sw* and the branching instructions.

The FSM works during both the posedge and negedge of the clock as follows:-

- **posedge** : Receives the decoded IR instruction and enables/disables ALU as per need, or updates PC in case of a branching instruction, or carries out the *lw* or *sw* instruction
- **negedge** : In-case of ALU instruction, the result is updated in the RD register during the negedge. This is done during negedge instead of posedge to make sure that the RD register doesn't accidently get assigned with an unknown value

  With this the processor can carry out each instruction within one cycle, and hence is ready to work on machine codes

PDS9-10: Writing the MIPs code for Bubble Sort and executing it on CSE-Bubble

The MIPs code converted and stored in the instruction memory of the Processor is as follows

```
     .data
arr: .word 10, 20, 15, 6, 8, 32, 78, 32
length: .word 8
space: .asciiz " "

.text
.globl main

main:
   jal print
   jal bubble
   li $v0, 10
   syscall

bubble:
   la $s0, arr              # load address of the array
   lw $s1, length           # load value of length of the array
   add $t0, $zero, $zero    # initialize value of i to 0
   addi $s7, $s1, -1    # $s7=length-1

loop1:
   slt $t8, $t0, $s7        # $t8=1 if i<length-1
   beq $t8, $zero, loop1exit #exit the 1st loop if t8=0
   add $t1, $zero, $zero    # initialize value of j to 0

loop2:
   sub $t9, $s7, $t0        # $t9=length-i-1
   slt $t8, $t1, $t9        # $t8=1 if j<length-i-1
   beq $t8, $zero, loop2exit # exit the 2nd loop if t8=0
   sll $t2, $t1, 2          # $t2=4*i=j
   add $t3, $t2, $s0        # $t3 stores current address arr[j]
   lw $s2, 0($t3)           # $s2 stores value of arr[j]
   addi $t4, $t3, 4        # $t4 stores address of arr[j+1]
   lw $s3, 0($t4)          # $s3 stores value of arr[j+1]
   slt $t5, $s3, $s2       # $t5=1 if arr[j+1]<arr[j]
   beq $t5, $zero, if   # exit if-else if $t5=0
   sw $s2, 0($t4)          # save value of arr[j] in arr[j+1]
   sw $s3, 0($t3)          # save value of arr[j+1] in arr[j]
if:
   addi $t1, $t1, 1        # j=j+1
   j loop2                 # restart the loop

loop2exit:
   addi $t0, $t0, 1        # i=i+1
   j loop1                 # restart the loop

loop1exit:

print:
print_loop_prep:
    la $t0, arr    # save address of array in $t0
lw $s1, length
add $t2, $zero, $zero  # initalize counter to print the array
print_loop:
bge $t2, $s1, print_end
li $v0, 1
lw $a0, 0($t0)
syscall
li $v0, 4
la $a0, space
syscall
addi $t0, $t0, 4
addi $t2, $t2, 1
j print_loop
print_end:
jr $ra
```

The print function is of course not implemented in the Processor code, but is there just for clarity and testing purposes