MIPS-ARCH-32

Problem Statement

The objective is to create a new processor CSE-BUBBLE that has the instruction set architecture as shown in Table 1 below. We assume that the processor word size and instruction size are 32 bits and we use the VEDA memory as implemented in Assignment 4. The VEDA memory has two parts: Instruction Memory & Data Memory

You have to make sure that these two sections do not overlap. We also assume that RISC-BUBBLE has total 32 registers of which certain registers follow the same roles as in MIPS-32 ISA. For example, Program Counter (PC) register holds the address of the next instruction. The target is to:

1. Create the op-code formats for the given ISA, shown in Table 1
2. Decide upon the data path elements (such as addition, subtraction, shift, jump)
3. Develop different verilog modules for the same
4. Develop the finite state machine for the control path
5. Build a single-cycle instruction execution unit for CSE-BUBBLE.

Table 1: Instruction Set for CSE-BUBBLE

Class	Instruction	Meaning
Arithmetic	add r0, r1, r2	r0=r1+r2
	sub r0, r1, r2	r0=r1-r2
	addu r0, r1, r2	r0=r1+r2 (unsigned addition, not 2's complement)
	subu r0,r1,r2	r0=r1-r2 (unsigned addition, not 2's complement)
	addi r0,r1,1000	r0=r1+1000
	addiu r0,r1, 1000	r0=r1+1000 (unsigned additon, not 2's complement)
Logical	and r0,r1,r2	r0= r1 & r2
	or r0,r1,r2	r0=r1 | r2
	andi r0,r1, 1000	r0= r1 & 1000
	ori r0,r1, 1000	r0= r1 | 1000
	sll r0, r1, 10	r0=r1<<10 (shift left logical)
	srl r0, r1, 10	r0=r1>>10 (shift right logical)
Data transfer	lw r0,10(r1)	r0=Memory[r1+10] (load word)
	sw r0,10(r1)	Memory[r1+10]=r0 (store word)
Conditional Branch	beq r0,r1,10	if(r0==r1) go to PC+4+10 (branch on equal)
	bne r0,r1,10	if(r0!=r1) go to PC+4+10 (branch on not equal)
	bgt r0,r1,10	if(r0>r1) go to PC+4+10 (branch if greater than)
	bgte r0,r1, 10	if(r0>=r1) go to PC+4+10 (branch if greter than or equal)
	ble r0,r1, 10	if(r0<r1) go to PC+4+10 (branch if less than)
	bleq r0,r1, 10	if(r0<=r1) go to PC+4+10 (branch if less than or equal)
Unconditional Branch	j 10	jump to address 10
	jr r0	jump to address stored in register r0
	jal 10	ra=PC+4 and jump to address 10
Comparison	slt r0,r1,r2	if(r1<r2) r0=1 else r0=0
	slti 1,2,100	if(r1<100) r0=1 else r0=0

Our Problem in hand is to create a hardware that can perform the bubble sort algorithm, given an array of numbers.

The algorithm itself can be summarized as:

```
 1. function bubbleSort(arr: list of sortable items):
 2.   n <= length(arr)
 3.   repeat:
 4.         swapped <= false
 5.         for i=1 to n-1 inclusive do
 6.               if arr[i-1] > arr[i] then
 7.                     swap(arr[i-1], arr[i])
 8.                     swapped <= true
 9.               end if
10.         end for
11.   until not swapped
12. end function
```

Clearly, the algorithm requires the capabilities of branching, adding, comparison, data transfer and logical operations.

MIPS program for BubbleSort can be:

```
.data

numbers: .word 8, 100, 0, 3, 7, 9, 2, 7, -3, 0  # create array which holds numbers
size: .word 10

.text
main:
    la      $s7,    numbers                     # load address of numbers into $s7
    li      $s0,    0                           # initialize counter 1 for loop 1
    lw      $s6,    size                        # n
    addi    $s6,    $s6,        -1              # n-1
    li      $s1,    0                           # initialize counter 2 for loop 2

loop:
    sll     $t7,    $s1,        2               # multiply $s1 by 2 and put it in t7
    add     $t7,    $s7,        $t7             # add the address of numbers to t7
    lw      $t0,    0($t7)                      # load numbers[j]
    lw      $t1,    4($t7)                      # load numbers[j+1]
    slt     $t2,    $t0,        $t1             # if t0 < t1
    bne     $t2,    $zero,      increment
    sw      $t1,    0($t7)                      # swap
    sw      $t0,    4($t7)

increment:
    addi    $s1,    $s1,        1               # increment t1
    sub     $s5,    $s6,        $s0             # subtract s0 from s6
    bne     $s1,    $s5,        loop
    addi    $s0,    $s0,        1               # otherwise add 1 to s0
    li      $s1,    0                           # reset s1 to 0
    bne     $s0,    $s6,        loop            # go back with s1 = s1 + 1
```

The above consists of 21 lines of instructions. Also the instructions used include:

`add, addi, sub, ori, sll, slt, sw, lw, bne, lui`

Pseudo instructions `li` and `la` are condensed to `ori` and `lui` repectively.

Processor Design Strategy

[PDS1] Decide registers and their usage protocols

Given the processor word size is 32b, and so is an instruction size. Therefore this would require 32b sized registers.

There can be 32 registers, with the following convention for being MIPS-compatible:

Number	Name	Usage
0	zero	constant zero
1	at	for assembler
2-3	v0-v1	for storing results
4-7	a0-a3	argument registers
8-15	t0-t7	temporary registers
16-23	s0-s7	saved registers
24-25	t8-t9	temporary registers, not for user
26-27	k0-k1	kernel registers
28	gp	global data pointer
29	sp	stack pointer
30	fp	frame pointer
31	ra	Return address

This convention would be helpful to extend the architecture's purpose for including furthur MIPS-compatible code.

[PDS2] Decide upon size of instruction and data memory in VEDA

Size of instruction and data memory depends on our purpose. Because this architecture is being formulated for implementation of `bubble_sort`, we can restrict their sizes accordingly.

We can decide upon keeping a maximum size of 2^{16} data memory, each address pointing to a word size, which would require 16b addresses. From above, 20 instructions were enough to implement the bubble sort, so keeping a buffer, size of 2^6 instruction memory would be sufficient.

[PDS3] Design instruction layout of '**R-**', '**I-**' and '**J-**' type instructions and their respective encoding methodologies

Again, keeping in mind the capability of executing MIPS code, we should keep instruction layout similar to MIPS.

Therefore R-type instruction would be of format:

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

Similarily, I-type instruction would be of format:

31-26	25-21	20-16	15-0
opcode	rs	rt	address/immediate
6b	**5b**	**5b**	**16b**

And J-type instruction format:

31-26	25-0
opcode	target address
6b	**26b**

[PDS4] Design and Implement an instruction fetch phase where the instruction next to the one executed will get stored in the instruction register

Implemented.

[PDS5] Design and implement a module for instruction decode to identify which data path element to execute given the opcode of the instruction

The following instructions need to be implemented, assume signed unless specified otherwise:

- R-type
  - Arithmetic: add, sub, addu, subu
  - Logical: and, or, sll, srl, **nor**
  - Comparison: slt
- I-type
  - Arithmetic: addi, addiu
  - Logical: andi, ori
  - Data Transfer: lw, sw

- Conditional: bne, beq
- Comparison: slti

➢ J-type

- Unconditional: j, jr, jal

Rest of the instructions are:

- Logical: not
- Data Transfer: li, la
- Conditional: bgt, bgte, ble, bleq
- Comparison: seq, sgt

These can be treated as pseudo instructions, breaking them down to:

`not $r0, $r1`	`nor $r0, $r1, $0`
`seq $r0, $r1, $r2`	`not $r2, $r2 ; invert $r2` `and $r0, $r1, $r2`
`sgt $r0, $r1, $r2`	`slt $r0, $r2, $r1`
`bgt $r0, $r1, LABEL`	`slt $t8, $r1, $r0` `bne $t8, $0, LABEL`
`bgte $r0, $r1, LABEL`	`slt $t8, $r0, $r1` `beq $t8, $0, LABEL`
`ble $r0, $r1, LABEL`	`slt $t8, $r0, $r1` `bne $t8, $0, LABEL`
`bleq $r0, $r1, LABEL`	`slt $t8, $r1, $r0` `beq $t8, $r0, LABEL`
`li $r0, VAL`	`ori $r0, $0, VAL`
`la $r0, ADDR`	`MANY FORMS`

The following are the encodings of the above needed instructions:

Instruction Opcodes:

\28-26b 31-29b	000	001	010	011	100	101	110	111
000	R-FORMAT		j	jal	beq	bne	blez	bgtz
001	addi	addiu	slti		andi	ori	xori	nori
010								
011								
100				lw				
101				sw				
110								

111								

if in R-FORMAT, Funct:

\2-0b 5-3b	000	001	010	011	100	101	110	111
000	sll		srl					
001	jr							
010								
011								
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu	seq			
110								
111								

[PDS6] Design and implement the Arithmetic Logic Unit (ALU) in top-down approach to develop different modules for different types of instructions. There might be some hardware parts in the ALU that can be shared by different modules if required. This will reduce the footprint of the hardware.

Note that if the instruction is of immediate type, but an operation, the ALU still assumes the value of immediate part is passed as $b.

ALU is required to implement the following operations:

Operations	ALU_CONTROL
`sll $a $b`	`000000`
`srl $a $b`	`000010`
`add $a $b`	`100000`
`addu $a $b`	`100001`
`sub $a $b`	`100010`
`subu $a $b`	`100011`
`and $a $b`	`100100`
`or $a $b`	`100101`
`xor $a $b`	`100110`
`nor $a $b`	`100111`
`slt $a $b`	`101010`
`sltu $a $b`	`101011`

[PDS7] Design and implement the branching operation along the ALU implementation.

Done.

[PDS8] Design the finite state machine for the control signals to execute the processor. Please ensure that every instruction should be executed in single clock cycle. Finally write the test benches to simulate the CSE-BUBBLE.

Done

[PDS9] Develop the MIPS code for Bubble Sort. Then convert it into machine code following the ISA given above.

Done

[PDS10] Store the machine code in the instruction memory and execute CSE-BUBBLE. Store the output of the bubble sort in the data memory.

Incomplete