Project Exam #2: Single Cycle MIPS Processor

Objective

In this project exam, you will implement and simulate a single-cycle MIPS processor. You will complete small but essential missing parts of the processor design to make it functional. **Follow the figure of the MIPS processor provided to guide your implementation.**

Your processor will be able to execute the following instructions at the end of this project exam:

- `add, sub, and, or, xor, nor, lw, sw, beq, addi`

Please advance step by step. Get help from the processor figure supplied in this sheet whenever needed. At the last page you will find MIPS Green Sheet for reference. At page 3, you will find the datapath for the MIPS processor.

Instructions

1. **Setup Your Project**

   ✓ Create a new project in Quartus.

   ✓ Copy the Verilog files to your project folder and Add them to the created project.

   ✓ Also copy the following memory initialization files to "yourprojectfolder/simulation/modelsim/" or "yourprojectfolder/simulation/questa/" as well. These mem files will be used to initialize the instruction memory, register and data memory respectively:

   1. data.mem
   2. reg.mem
   3. instructions.mem

   instructions.mem content is as follows:

```
00000000000000000000000000000000 // NOP
00000001000010011000000000100000 // ADD  $s0, $t0, $t1
00000001000010011000100000100010 // SUB  $s1, $t0, $t1
00000001000010011001000000100100 // AND  $s2, $t0, $t1
00000001000010011001100000100101 // OR   $s3, $t0, $t1
00000001000010011010000000100110 // XOR  $s4, $t0, $t1
00000001000010011010100000100111 // NOR  $s5, $t0, $t1
10101100000010000000000000001000 // SW   $t0, 8($zero)
10001100000010100000000000001000 // LW   $t2, 8($zero)
00010001000010010000000000000001 // BEQ  $t1, $t0, 1
00100000000010110000000000010111 // ADDI $t3, $zero, 23
00100000000011000000000000011001 // ADDI $t4, $zero, 25
00100000000011010000000000011011 // ADDI $t5, $zero, 27
```

   ✓ Set the **SingleCycleMIPS.v** file as the **Top-Level Module** in your project.

- ✓ You also have a testbench as **Tb.v** which executes the instructions in instructions.mem and write the resultant register and data memory contents to a file under "**yourprojectfolder/simulation/modelsim/**" or "**yourprojectfolder/simulation/questa/**" as **testbench_output.txt**. In order to obtain that file, you must simulate your project using **Tb.v**.

2. **Compile the Project**
   - ✓ Compile the project to ensure all files are properly linked and there are no syntax errors.

Implementation Tasks

The "**SingleCycleMIPS.v**" file provided contains commented sections where additional components need to be implemented. Follow the steps below to complete the processor:

Part A: Add Multiplexers

- ☞ Implement the multiplexers of the processor as shown in the provided MIPS processor figure.
- ☞ You can implement "`assign next_pc = pc + 4`" in the commented-out section in "**SingleCycleMIPS.v**" to succesfully complete Part A. Because you will not execute `beq` in Part A.

Part B: Program Counter for Branch Operations

- ☞ Change "**`next_pc`**" assignment so that the processor can support **`beq`**.
- ☞ You can see clearly how you must implement "**`next_pc`**" logic for branch operations in the datapath figure.

Part C: Implement ADDI Instruction

For this part you must modify "**ControlUnit.v**".

In MIPS processors, the control unit generates the necessary signals for each instruction to ensure the processor functions correctly. In this section, you are expected to make the necessary adjustments in the control unit to implement the **`addi`** (Add Immediate) instruction.

- ☞ **About the Control Unit:**
  The control unit determines which signals the processor will work with. It uses the opcode of a given instruction to generate control signals.
- ☞ **For the `addi` Instruction:**
  Determine which values the signals like **RegDst**, **Branch**, **MemRead**, and **MemtoReg** should take to execute this instruction. For example: **reg_write**: Controls whether the result will be written to the register.
- ☞ **`addi` Instruction:**
  The opcode value for the **`addi`** instruction is **`6'b001000`**.
  This instruction adds an immediate (constant) value to a register's value and stores the result in a destination register.
- ☞ **General Format:**
  `addi rt, rs, immediate (R[rt] <- R[rs] + sign_ext_imm)`

Where:

rs: Source register

rt: Destination register

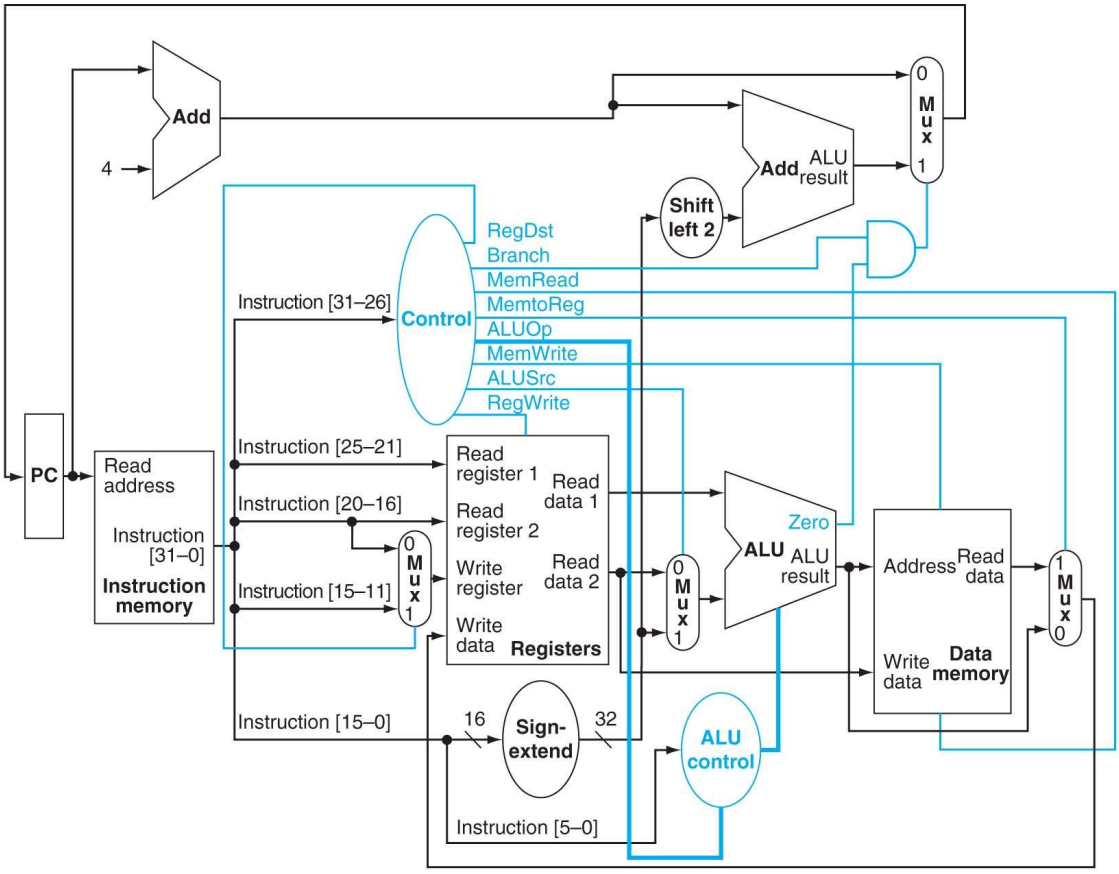
immediate: Constant value (signed 16-bit)

☞ **Explain Your Code:**

Add a brief explanation of the case you implemented, describing how you set the control signals and how these signals affect the processor's operation.

Submission Guidelines

1. Compile and simulate your Project, your simulation results will be written in a file called "**testbench_output.txt**" automatically.
2. Make sure you wrote some small comment to explain what you did for your code.
3. Compress and upload your Project to the assignment section in the Team.
4. If you did not create a Project and compile it successfully, your code will not be evaluated.



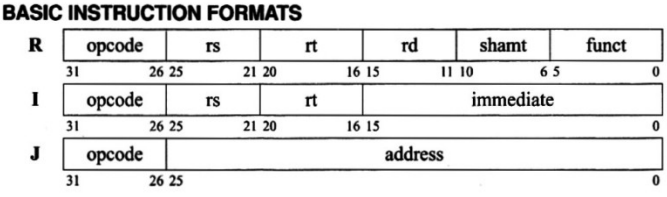
MIPS Reference Data

CORE INSTRUCTION SET

NAME	MNEMONIC	FORMAT	OPERATION (in Verilog)		OPCODE/ FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1)(2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr*4	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr*4	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+4;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2)(6)	b_{hex}
Set Less Than Unsigned	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2 s comp.)

BASIC INSTRUCTION FORMATS



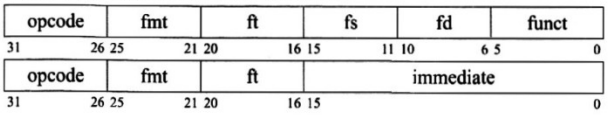
ARITHMETIC CORE INSTRUCTION SET

NAME	MNEMONIC	FORMAT	OPERATION		OPCODE/ FMT / FT/ FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		16 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING POINT INSTRUCTION FORMATS



PSEUDO INSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

Copyright 2005 by Elsevier, Inc., All rights reserved. From Patterson and Hennessy, *Computer Organization and Design,* 3rd ed.