THE UNIVERSITY OF NEWCASTLE AUSTRALIA

School of Engineering

Dr Peter Stepien (peter.stepien@newcastle.edu.au)

Assignment 2

Instructions

- Complete all questions.
- Work in groups of two.
- Assignment files due end of Week 12 on Friday 24 October 2025 by 11:59pm.
- Demonstration during laboratory session in Week 13.
- Total of 80 Marks possible.

Assessment

This assignment will be assessed using the criteria in Table 1 for each of the questions:

Component	*Unsatisfactory*	*Improving*	*Satisfactory*	*Excellent*
Question 1 (30 Marks)	Processor not working	Simulating, but not on FPGA	All working with good comments	
Question 2 (10 Marks)	Immediate instructions not working	Simulating, but not on FPGA	All working with good comments	Additional immediate instructions implemented
Question 3 (10 Marks)	Branch instruction not working	Simulating, but not on FPGA	All working with good comments	Additional branch instructions implemented
Question 4 (10 Marks)	Memory instructions not working	Simulating, but not on FPGA	All working with good comments	
Question 5 (10 Marks)	Jump instruction not working	Simulating, but not on FPGA	Jump working with good comments	

Table 1: Marking criteria

Objectives

The aim of this assignment is to design a 32-bit single-cycle MIPS processor which supports the subset of the instruction set shown in Table 2. A quick reference for the MIPS instruction set is given in Figure 4.

Blocks created during the first assignment will be used as the basis for the design in the second assignment. Complete the assignment one question at a time, implementing additional instructions each time.

Refer to the lecture slides for Topics 10 and 11 and Chapter 7 of the textbook for information about the MIPS architecture that will help you complete the assignment.

Note that both structural and behavioural designs are accepted. The designs for all questions must be accompanied with test benches that are tested in ModelSim and hardware implementations that will run on the FPGA development board.

Table 2: Subset of MIPS instruction set to be implemented

Instruction	*Syntax*	*Encoding*
Add	`add $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 0000`
Add unsigned	`addu $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 0001`
Subtract	`sub $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 0010`
Subtract unsigned	`subu $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 0011`
Bitwise AND	`and $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 0100`
Bitwise OR	`or $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 0101`
Bitwise XOR	`xor $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 0110`
Bitwise NOR	`nor $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 0111`
Set on less than (signed)	`slt $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 1010`
Set on less than (unsigned)	`sltu $d, $s, $t`	`0000 00ss ssst tttt dddd d000 0010 1011`
Jump	`j target`	`0000 10ii iiii iiii iiii iiii iiii iiii`
Branch on equal	`beq $s, $t, offset`	`0001 00ss ssst tttt iiii iiii iiii iiii`
Add immediate	`addi $t, $s, imm`	`0010 00ss ssst tttt iiii iiii iiii iiii`
Add immediate unsigned	`addiu $t, $s, imm`	`0010 01ss ssst tttt iiii iiii iiii iiii`
Load word	`lw $t, offset($s)`	`1000 11ss ssst tttt iiii iiii iiii iiii`
Store word	`sw $t, offset($s)`	`1010 11ss ssst tttt iiii iiii iiii iiii`

Switches and LEDs

Use the switches and LEDs to be able to view the program counter and registers. Use the switches and LEDs as shown in Figure 1. The switches select which register to view. Only 16-bits of the registers can be viewed at a time. A push button should be used for the clock signal. Use the other LEDs for debugging as you want.

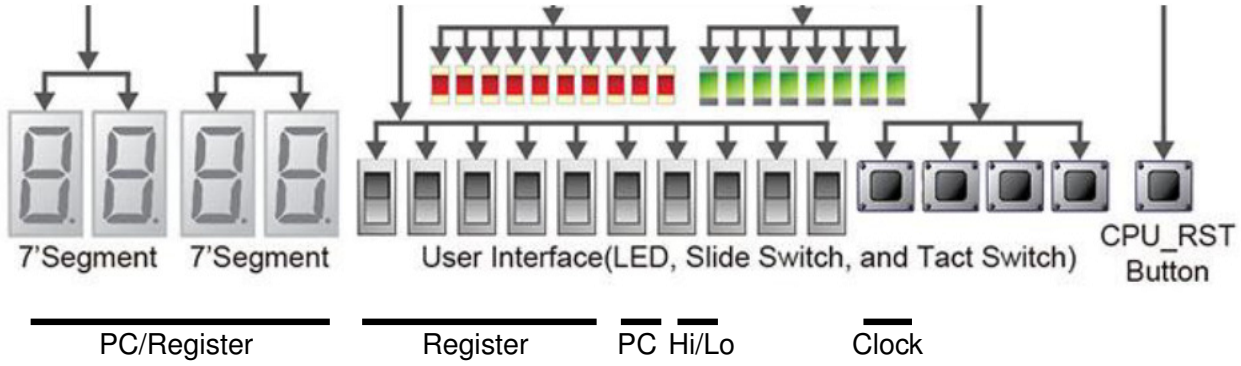


Figure 1: Mapping of indicators and switches

Submission

The complete Quartus project is to be submitted on Canvas in a single ZIP file. Include all test benches used for each question and documentation of your design within the System Verilog source files. The solutions will also need to be demonstrated during the normal scheduled laboratory times.

1 Register Instructions (30 Marks)

Design a basic 32-bit single-cycle MIPS processor which comprises instruction memory, a register file, ALU and control logic as shown in Figure 2 but without the data memory. The design should be able to execute the R-type instructions listed at the start of Table 2.

Hint: Refer to Assignment 1 Question 9 for the ALU hardware design. The control logic requires the instruction in Assignment 1 Question 6.

2 Immediate Instructions (10 Marks)

Update the control logic to allow the execution of the two I-type instructions listed in Table 2. Feel free to implement additional immediate instructions that do not require further changes to the data path.

Hint: The control logic uses the MUX to route the immediate operand to the ALU.

3 Branch Instruction (10 Marks)

Implement the branch instruction, `beq`, to allow conditional branches. Test the operation using a loop.

Hint: Write a program that will loop a predetermined number of times and then exit to test your design.

4 Memory Instructions (10 Marks)

Complete the complete design in Figure 2 by adding the data memory. Implement the memory load and store instructions, `lw` and `sw` repectively. Test correct operation by writing and reading to data memory.

Hint: The MUX is used to select the between the output of data memory or the ALU.

5 Jump Instruction (10 Marks)

Implement the jump instruction, `j`. This requires additional circuit as shown Figure 3.

Hint: Refer to lecture slides for Topic 11.

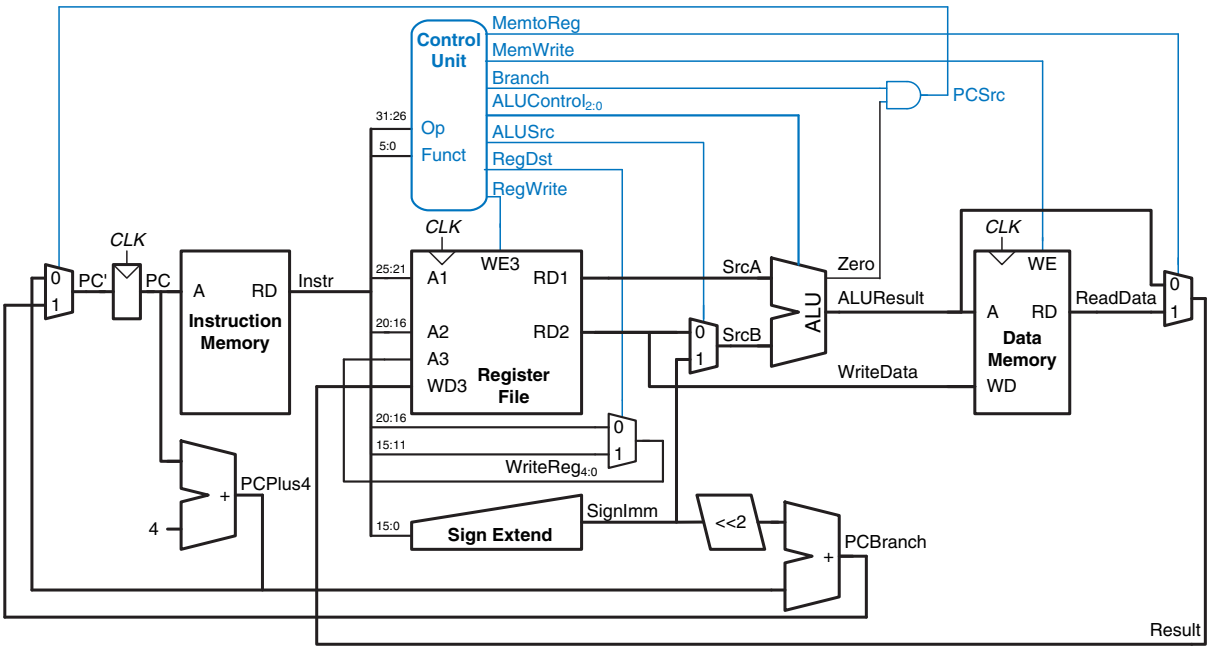


Figure 2: 32-bit single-cycle MIPS processor

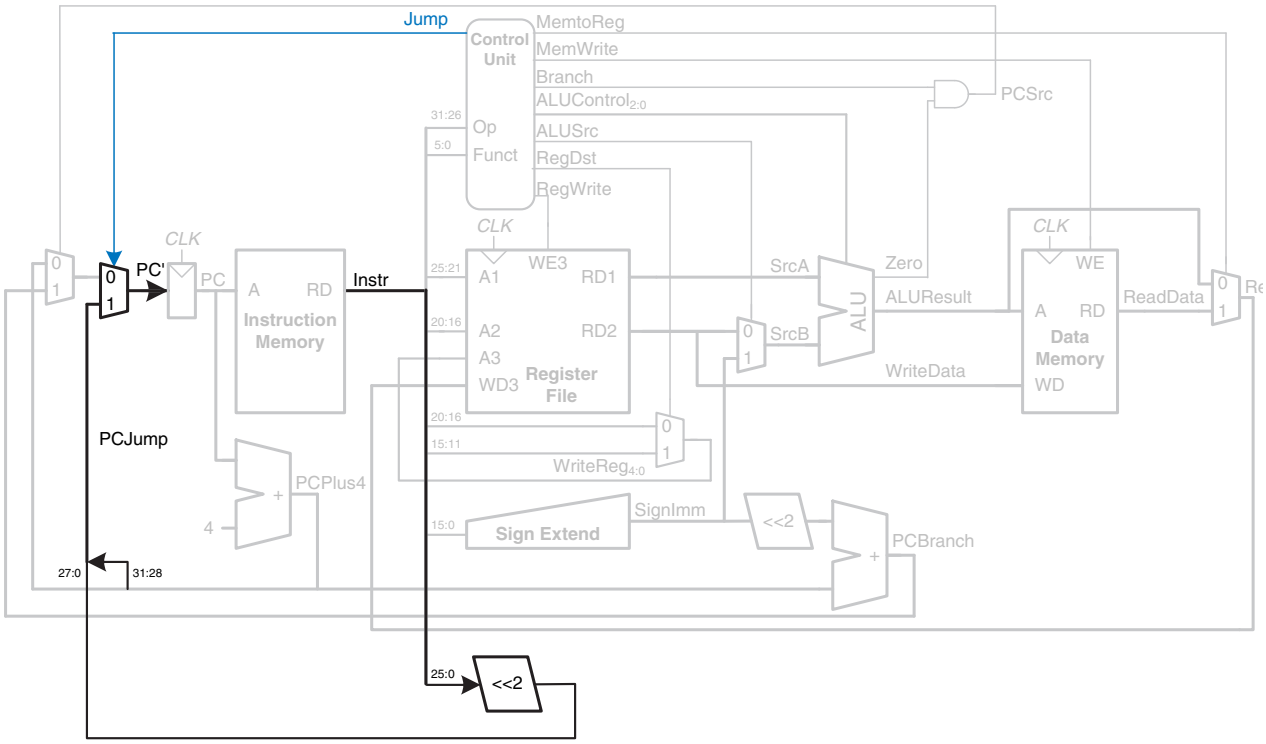


Figure 3: Additional circuit for jump instruction

MIPS Reference Sheet

David Broman, KTH Royal Institute of Technology
Version 1.12, November 6, 2015

INSTRUCTIONS (SUBSET)

Name (format, op, funct)	Syntax	Operation
add (R,0,32)	`add rd,rs,rt`	reg(rd) := reg(rs) + reg(rt);
add immediate (I,8,na)	`addi rt,rs,imm`	reg(rt) := reg(rs) + signext(imm);
add immediate unsigned (I,9,na)	`addiu rt,rs,imm`	reg(rt) := reg(rs) + signext(imm);
add unsigned (R,0,33)	`addu rd,rs,rt`	reg(rd) := reg(rs) + reg(rt);
and (R,0,36)	`and rd,rs,rt`	reg(rd) := reg(rs) & reg(rt);
and immediate (I,12,na)	`andi rt,rs,imm`	reg(rt) := reg(rs) & zeroext(imm);
branch on equal (I,4,na)	`beq rs,rt,label`	if reg(rs) == reg(rt) then PC = BTA else NOP;
branch on not equal (I,5,na)	`bne rs,rt,label`	if reg(rs) != reg(rt) then PC = BTA else NOP;
jump and link register (R,0,9)	`jalr rs`	$ra := PC + 4; PC := reg(rs);
jump register (R,0,8)	`jr rs`	PC := reg(rs);
jump (J,2,na)	`j label`	PC := JTA;
jump and link (J,3,na)	`jal label`	$ra := PC + 4; PC := JTA;
load byte (I,32,na)	`lb rt,imm(rs)`	reg(rt) := signext(mem[reg(rs) + signext(imm)]$_{7:0}$);
load byte unsigned (I,36,na)	`lbu rt,imm(rs)`	reg(rt) := zeroext(mem[reg(rs) + signext(imm)]$_{7:0}$);
load upper immediate (I,15,na)	`lui rt,imm`	reg(rt) := concat(imm, 16 bits of 0);
load word (I,35,na)	`lw rt,imm(rs)`	reg(rt) := mem[reg(rs) + signext(imm)];
multiply, 32-bit result (R,28,2)	`mul rd,rs,rt`	reg(rd) := reg(rs) * reg(rt);
nor (R,0,39)	`nor rd,rs,rt`	reg(rd) := not(reg(rs) | reg(rt));
or (R,0,37)	`or rd,rs,rt`	reg(rd) := reg(rs) | reg(rt);
or immediate (I,13,na)	`ori rt,rs,imm`	reg(rt) := reg(rs) | zeroext(imm);
set less than (R,0,42)	`slt rd,rs,rt`	reg(rd) := if reg(rs) < reg(rt) then 1 else 0;
set less than unsigned (R,0,43)	`sltu rd,rs,rt`	reg(rd) := if reg(rs) < reg(rt) then 1 else 0;
set less than immediate (I,10,na)	`slti rt,rs,imm`	reg(rt) := if reg(rs) < signext(imm) then 1 else 0;
set less than immediate unsigned (I,11,na)	`sltiu rt,rs,imm`	reg(rt) := if reg(rs) < signext(imm) then 1 else 0; (inequality < compares using unsigned values)
shift left logical (R,0,0)	`sll rd,rt,shamt`	reg(rd) := reg(rt) << shamt;
shift left logical variable (R,0,4)	`sllv rd,rt,rs`	reg(rd) := reg(rt) << reg(rs$_{4:0}$);
shift right arithmetic (R,0,3)	`sra rd,rt,shamt`	reg(rd) := reg(rt) >>> shamt;
shift right logical (R,0,2)	`srl rd,rt,shamt`	reg(rd) := reg(rt) >> shamt;
shift right logical variable (R,0,6)	`srlv rd,rt,rs`	reg(rd) := reg(rt) >> reg(rs$_{4:0}$);
store byte (I,40,na)	`sb rt,imm(rs)`	mem[reg(rs) + signext(imm)]$_{7:0}$:= reg(rt)$_{7:0}$;
store word (I,43,na)	`sw rt,imm(rs)`	mem[reg(rs) + signext(imm)] := reg(rt);
subtract (R,0,34)	`sub rd,rs,rt`	reg(rd) := reg(rs) - reg(rt);
subtract unsigned (R,0,35)	`subu rd,rs,rt`	reg(rd) := reg(rs) - reg(rt);
xor (R,0,38)	`xor rd,rs,rt`	reg(rd) := reg(rs) ^ reg(rt);
xor immediate (I,14,na)	`xori rt,rs,imm`	reg(rt) := rerg(rs) ^ zeroext(imm);

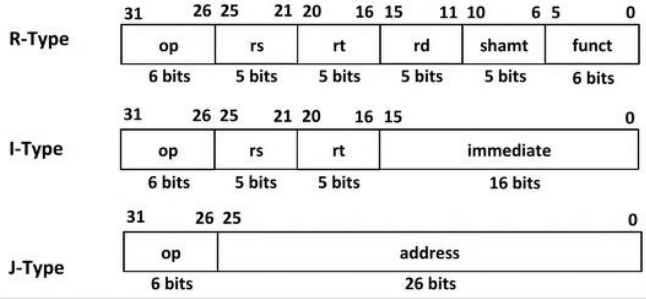
PSEUDO INSTRUCTIONS (SUBSET)

Name	Example	Equivalent Basic Instructions
load address	`la $t0,label`	`lui $at,hi-bits-of-address` `ori $t0,$at,lower-bits-of-address`
load immediate	`li $t0,0xabcd1234`	`lui $at,0xabcd` `ori $t0,$at,0x1234`
branch if less or equal	`ble $t0,$t1,label`	`slt $at,$t1,$t0` `beq $at,$zero,label`
move	`move $t0,$t1`	`addi $t0,$t1,$zero`
no operation	`nop`	`sll $zero,$zero,0`

ASSEMBLER DIRECTIVES (SUBSET)

data section	`.data`
ASCII string declaration	`.ascii "a string"`
word alignment	`.align 2`
word value declaration	`.word 99`
byte value declaration	`.byte 7`
global declaration	`.global foo`
allocate X bytes of space	`.space X`
code section	`.text`

INSTRUCTION FORMAT



REGISTERS

Name	Number	Description
$0, $zero	0	constant value 0
$at	1	assembler temp
$v0	2	function return
$v1	3	function return
$a0	4	argument
$a1	5	argument
$a2	6	argument
$a3	7	argument
$t0	8	temporary value
$t1	9	temporary value
$t2	10	temporary value
$t3	11	temporary value
$t4	12	temporary value
$t5	13	temporary value
$t6	14	temporary value
$t7	15	temporary value
$s0	16	saved temporary
$s1	17	saved temporary
$s2	18	saved temporary
$s3	19	saved temporary
$s4	20	saved temporary
$s5	21	saved temporary
$s6	22	saved temporary
$s7	23	saved temporary
$t8	24	temporary value
$t9	25	temporary value
$k0	26	reserved for OS
$k1	27	reserved for OS
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Definitions

- Jump to target address: JTA = concat((PC + 4)$_{31:28}$, address(label), 00_2)
- Branch target address: BTA = PC + 4 + signext(imm) * 4

Clarifications

- All numbers are given in decimal form (base 10).
- Function signext(x) returns a 32-bit sign extended value of x in two's complement form.
- Function zeroext(x) returns a 32-bit value, where zero are added to the most significant side of x.
- Function concat(x, y, ..., z) concatenates the bits of expressions x, y, ..., z.
- Subscripts, for instance X$_{8:2}$, means that bits with index 8 to 2 are spliced out of the integer X.
- Function *address*(x) means the address of label x.
- NOP and na mean "no operation" and "not applicable", respectively.
- shamt is an abbreviation for "shift amount", i.e. how many bits that should be shifted.
- `addu` and `addiu` are misnamed *unsigned* because an add operation handles both signed and unsigned numbers in the same way. The term unsigned is actually used to describe that the instruction does not throw overflow exceptions.

Figure 4: MIPS Reference Sheet