Laboratory 04

Single-Cycle MIPS CPU Design – smaller: 16-bits version

One clock cycle per instruction

1. Objectives

Study, design, implement and test

- **Single-Cycle MIPS CPU**

Familiarize the students with

- Single-Cycle CPU design: Defining the instructions / writing the test program (MIPS assembly language, machine code)
- Xilinx® ISE Webpack
- Digilent Development Boards (DDB)
  - **Digilent Basys Board – Reference Manual**

2. Reduced Size MIPS Processor Description

(!) Read Lectures 3 and 4 in order to understand the works needed in this laboratory.

In this laboratory you will design and start the implementation of your own single cycle MIPS processor – MIPS 16.

The microprocessor will be a simpler version of the MIPS 32 microarchitecture described during the lectures. What does simpler mean? The instruction set will be smaller (fewer instructions to implement); the width of the instructions and data fields will be of 16-bits. Implicitly, the number of registers used in the register file will be smaller; the instruction and data memories will be smaller. The rest of the principles described during the lectures are the same (data-path and control).

The main reason for implementing MIPS 16 is the reduced methodologies for data display (8 or 16 LEDs and 4-digit Seven Segment Display). In this manner one avoids using other multiplexing mechanisms for signal display purposes (32 bits); and the on-chip debugging process is simplified (testing your program on the FPGA board).

The dimension/width of both instructions and data will be of 16-bits. The 3 instruction formats are given below. Compare this format with the 32-bit instruction format from the lectures. Observe the differences/limitations.

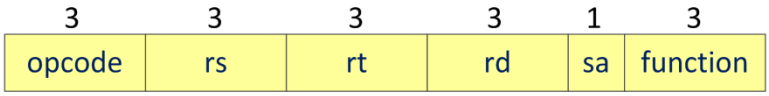

Figure 1: R-type Instruction format

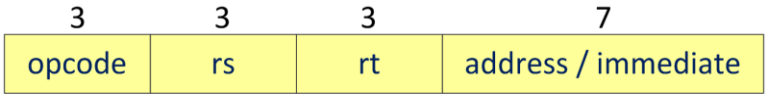

Figure 2: I-type Instruction format

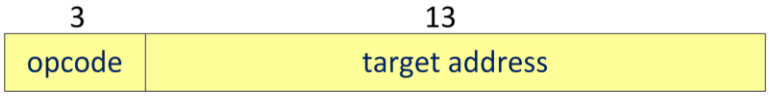

Figure 3: J-type Instruction format

These instruction formats obey the formats presented in the MIPS32 ISA, except the width of each field.

The opcode is encoded on 3-bits. For I-type and J-type instructions, the opcode uniquely encodes the instruction to be executed. In the case of R-type instructions, in accordance to the MIPS standard, the opcode is 0 and the function field identifies the ALU operation for each instruction. The function field is encoded on 3-bits. This means that your processor can implement at most 15 instructions:

- 8 R-type Instructions
- 7 I-type Instructions and J-type instructions.

The table below presents the minimum number of instructions, of each type, that will be implemented on the MIPS 16 processor. On the doted positions you will choose or define new instructions for your MIPS processor (depending on the program that you will implement).

R-type Instructions	Addition	add
	Subtraction	sub
	Shift Left Logical (with shift amount – sa)	sll
	Shift Right Logical (with shift amount – sa)	srl
	Logical AND	and
	Logical OR	or
	….	…
	….	…
I-type Instructions	Add Immediate	addi
	Load Word	lw
	Store Word	sw
	Branch on Equal	beq
	….	…
	….	…
J-type Instruction	Jump	j

Table 1: Instructions for MIPS16

The description of each MIPS 16 data-path component characteristics is given below. (!) These characteristics are valid not only for this laboratory, but also for the future laboratory works.

Program Counter characteristics:

- 16-bit edge triggered D flip-flop

Instruction Memory (ROM) characteristics:

- One input bus: Instruction Address
- One output bus: Instruction Data
- Memory word is 16-bit (selected by instruction address)
- No control signals

Register File characteristics:

- Two read addresses and one write address
- Eight 16-bit registers (rs, rt, rd encoded on 3-bits)
- Two 16-bit data outputs: Read data 1 and Read data 2
- One 16-bit data input: Write Data
- Multiple accesses: 2 asynchronous reads and 1 synchronous (edge triggered) write. During read operation the register file behaves as a combinational logic block.
- One control signal RegWrite. When RegWrite is asserted the value on the Write Data line is written in the register indicated by the write address line

Data Memory (RAM) characteristics:

- One 16-bit input address bus: Address
- One 16-bit input data bus: Write Data
- One 16-bit output data bus: Read Data
- One control Signal: MemWrite

Extension Unit characteristics:

- ExtOp = 1 → Sign Extension
- ExtOp = 0 → Zero Extension

ALU characteristics:

- ALU performs arithmetical / logical operations
- (!) You need to identify all the operations that the ALU needs to perform after completing the definition of the instructions from Table 1. You are encouraged to choose two more R-type instructions and two more I-type Instructions.
- You need to identify how many control bits are necessary to encode the ALU operations (ALUCtrl).

3. Laboratory Assignments

Read carefully and completely each activity before you begin!

3.1. Define the instructions for MIPS 16 – Paper and Pencil

Starting from the instruction formats presented in the previous section write (Paper and Pencil) the instruction format (on bits, including the opcode and function fields) for each of the instructions presented in Table 1.

Add two more R-type instructions and 2 more I-type instructions in order to complete the whole number of instructions that your processor is capable of performing.

Besides the lecture materials you can also use Appendix 5 for a reference on MIPS instructions.

You need to specify the encoding (on bits) in all of the fields from the instruction.

Write the RTL abstract for all the 15 instructions from your MIPS 16 instruction set. Draw the processing diagram for all the instructions (add, and, sll, lw, beq, j – in the laboratory, the rest as homework).

For your MIPS 16 processor you will ignore the overflow exceptions that can appear during ALU operations (example: add instruction)

Give an example for each instruction including the bit encoding off all fields (including the instruction operands). Example: add $2, $4, $3 → "… the 16 bits…".

Attention: in order to increase the encoding readability of each instruction use the "_" symbol between the instruction fields (opcode, rs, etc.). This is also supported by VHDL and has no effect on the bit string. For VHDL it is mandatory to specify the binary encoding B before the string of bits (or X, O for hexadecimal and octal encoding respectively).

```
B"001_010_011_100_1_111" is equivalent to "0010100111001111"
```

3.2. MIPS 16 test program

Write a short program with the instructions that you have defined for your MISP 16 processor (paper and pencil). Describe the program in assembly language, then each instruction in machine code (16-bit encoding for each instruction, use "_" between the fields of the instructions).

Using assignment 3.1 from laboratory 3 (ROM memory whose addresses are generated by a Mono Pulse Generator controlled counter), introduce your program

in the ROM memory and trace it on the Basys board. When writing your program in the ROM memory you have to write a comment for each instruction, i.e. the assembly language description for each instruction. Your program should be visible in parallel to the machine code.

Optionally, you are invited to write a more complex program for your MIPS processor.

3.3. Data-Path for MIPS 16

Draw the Data-Path of your single-cycle MIPS 16 processor. Be sure to include all the necessary components on the data-path, such that all the 15 instructions execute correctly.

Starting from the RTL abstract description, identify the values of the control signals for each instruction. Draw a table with the control signals and their values (see lecture 4 for details).

4. References

- Computer Architecture Lectures 3 & 4 slides.
- MIPS® Architecture For Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
- MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
- MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
    - Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Appendix 5 – MIPS Instruction Reference

Note: ALL immediate values should be sign extended.
Exception: For logical operations immediate values should be zero extended.
After extensions, you treat them as signed or unsigned 32 bit numbers.

For the non-immediate instructions, the only difference between signed and unsigned instructions (ex ADD vs. ADDU) is that signed instructions can generate an overflow.

The instruction formats are given, you can figure out the binary instruction codes. The instruction descriptions are given below. Additional details can be found here: "MIPS Single Cycle Processor", John Alexander, Barret Schloerke, Daniel Sedam, Iowa State University

ADD – Add

Description:	Adds two registers and stores the result in a register
Operation:	$d ← $s + $t; advance_pc (4);
Syntax:	add $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0000

ADDI – Add immediate

Description:	Adds a register and a signed immediate value and stores the result in a register
Operation:	$t ← $s + imm; advance_pc (4);
Syntax:	addi $t, $s, imm
Encoding:	0010 00ss ssst tttt iiii iiii iiii iiii

ADDIU – Add immediate unsigned

Description:	Adds a register and an unsigned immediate value and stores the result in a register
Operation:	$t ← $s + imm; advance_pc (4);
Syntax:	addiu $t, $s, imm
Encoding:	0010 01ss ssst tttt iiii iiii iiii iiii

ADDU – Add unsigned

Description:	Adds two registers and stores the result in a register
Operation:	$d ← $s + $t; advance_pc (4);
Syntax:	addu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0001

AND – Bitwise and

Description:	Bitwise ands two registers and stores the result in a register
Operation:	$d ← $s & $t; advance_pc (4);
Syntax:	and $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0100

ANDI – Bitwise and immediate

Description:	Bitwise ands a register and an immediate value and stores the result in a register
Operation:	$t ← $s & imm; advance_pc (4);
Syntax:	andi $t, $s, imm
Encoding:	0011 00ss ssst tttt iiii iiii iiii iiii

BEQ – Branch on equal

Description:	Branches if the two registers are equal
Operation:	if $s == $t advance_pc (offset << 2); else advance_pc (4);
Syntax:	beq $s, $t, offset
Encoding:	0001 00ss ssst tttt iiii iiii iiii iiii

BGEZ – Branch on greater than or equal to zero

Description:	Branches if the register is greater than or equal to zero
Operation:	if $s >= 0 advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgez $s, offset
Encoding:	0000 01ss sss0 0001 iiii iiii iiii iiii

BGEZAL – Branch on greater than or equal to zero and link

Description:	Branches if the register is greater than or equal to zero and saves the return address in $31
Operation:	if $s >= 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgezal $s, offset
Encoding:	0000 01ss sss1 0001 iiii iiii iiii iiii

BGTZ – Branch on greater than zero

Description:	Branches if the register is greater than zero
Operation:	if $s > 0 advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgtz $s, offset
Encoding:	0001 11ss sss0 0000 iiii iiii iiii iiii

BLEZ – Branch on less than or equal to zero

Description:	Branches if the register is less than or equal to zero
Operation:	if $s <= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	blez $s, offset
Encoding:	0001 10ss sss0 0000 iiii iiii iiii iiii

BLTZ – Branch on less than zero

Description:	Branches if the register is less than zero
Operation:	if $s < 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltz $s, offset
Encoding:	0000 01ss sss0 0000 iiii iiii iiii iiii

BLTZAL – Branch on less than zero and link

Description:	Branches if the register is less than zero and saves the return address in $31
Operation:	if $s < 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltzal $s, offset
Encoding:	0000 01ss sss1 0000 iiii iiii iiii iiii

BNE – Branch on not equal

Description:	Branches if the two registers are not equal
Operation:	if $s != $t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bne $s, $t, offset
Encoding:	0001 01ss ssst tttt iiii iiii iiii iiii

DIV – Divide

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO ← $s / $t; $HI ← $s % $t; advance_pc (4);
Syntax:	div $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1010

DIVU – Divide unsigned

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO ← $s / $t; $HI ← $s % $t; advance_pc (4);
Syntax:	divu $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1011

J – Jump

Description:	Jumps to the calculated address
Operation:	PC ← nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	j target
Encoding:	0000 10ii iiii iiii iiii iiii iiii iiii

JAL – Jump and link

Description:	Jumps to the calculated address and stores the return address in $31
Operation:	$31 ← PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	jal target
Encoding:	0000 11ii iiii iiii iiii iiii iiii iiii

JR – Jump register

Description:	Jump to the address contained in register $s
Operation:	PC ← nPC; nPC = $s;
Syntax:	jr $s
Encoding:	0000 00ss sss0 0000 0000 0000 0000 1000

LB – Load byte

Description:	A byte is loaded into a register from the specified address.
Operation:	$t ← MEM[$s + offset]; advance_pc (4);
Syntax:	lb $t, offset($s)
Encoding:	1000 00ss ssst tttt iiii iiii iiii iiii

LUI – Load upper immediate

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	$t ← (imm << 16); advance_pc (4);
Syntax:	lui $t, imm
Encoding:	0011 11-- ---t tttt iiii iiii iiii iiii

LW – Load word

Description:	A word is loaded into a register from the specified address.
Operation:	$t ← MEM[$s + offset]; advance_pc (4);
Syntax:	lw $t, offset($s)
Encoding:	1000 11ss ssst tttt iiii iiii iiii iiii

MFHI – Move from HI

Description:	The contents of register HI are moved to the specified register.
Operation:	$d ← $HI; advance_pc (4);
Syntax:	mfhi $d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0000

MFLO – Move from LO

Description:	The contents of register LO are moved to the specified register.
Operation:	$d ← $LO; advance_pc (4);
Syntax:	mflo $d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0010

MULT – Multiply

Description:	Multiplies $s by $t and stores the result in $Hi and $LO.
Operation:	$Hi, $LO ← $s * $t; advance_pc (4);
Syntax:	mult $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1000

MULTU – Multiply unsigned

Description:	Multiplies $s by $t and stores the result in $Hi and $LO.
Operation:	$Hi, $LO ← $s * $t; advance_pc (4);
Syntax:	multu $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1001

NOOP – no operation

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	0000 0000 0000 0000 0000 0000 0000 0000

Note: The encoding for a NOOP represents the instruction SLL $0, $0, 0 which has no side effects. In fact, nearly every instruction that has $0 as its destination register will have no side effect and can thus be considered a NOOP instruction.

OR – Bitwise or

Description:	Bitwise logical ors two registers and stores the result in a register
Operation:	$d ← $s | $t; advance_pc (4);
Syntax:	or $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0101

ORI – Bitwise or immediate

Description:	Bitwise ors a register and an immediate value and stores the result in a register
Operation:	$t ← $s | imm; advance_pc (4);
Syntax:	ori $t, $s, imm
Encoding:	0011 01ss ssst tttt iiii iiii iiii iiii

SB – Store byte

Description:	The least significant byte of $t is stored at the specified address.
Operation:	MEM[$s + offset] ← (0xff & $t); advance_pc (4);
Syntax:	sb $t, offset($s)
Encoding:	1010 00ss ssst tttt iiii iiii iiii iiii

SLL – Shift left logical

Description:	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation:	$d ← $t << h; advance_pc (4);
Syntax:	sll $d, $t, h
Encoding:	0000 00ss ssst tttt dddd dhhh hh00 0000

SLLV – Shift left logical variable

Description:	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation:	$d ← $t << $s; advance_pc (4);
Syntax:	sllv $d, $t, $s
Encoding:	0000 00ss ssst tttt dddd d--- --00 0100

SLT – Set on less than (signed)

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d ← 1; advance_pc (4); else $d ← 0; advance_pc (4);
Syntax:	slt $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1010

SLTI – Set on less than immediate (signed)

Description:	If $s is less than immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t ← 1; advance_pc (4); else $t ← 0; advance_pc (4);
Syntax:	slti $t, $s, imm
Encoding:	0010 10ss ssst tttt iiii iiii iiii iiii

SLTIU – Set on less than immediate unsigned

Description:	If $s is less than the unsigned immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t ← 1; advance_pc (4); else $t ← 0; advance_pc (4);
Syntax:	sltiu $t, $s, imm
Encoding:	0010 11ss ssst tttt iiii iiii iiii iiii

SLTU – Set on less than unsigned

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d ← 1; advance_pc (4); else $d ← 0; advance_pc (4);
Syntax:	sltu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1011

SRA – Shift right arithmetic

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation:	$d ← $t >> h; advance_pc (4);
Syntax:	sra $d, $t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0011

SRL – Shift right logical

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation:	$d ← $t >> h; advance_pc (4);
Syntax:	srl $d, $t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0010

SRLV – Shift right logical variable

Description:	Shifts a register value right by the amount specified in $s and places the value in the destination register. Zeroes are shifted in.
Operation:	$d ← $t >> $s; advance_pc (4);
Syntax:	srlv $d, $t, $s
Encoding:	0000 00ss ssst tttt dddd d000 0000 0110

SUB – Subtract

Description:	Subtracts two registers and stores the result in a register
Operation:	$d ← $s - $t; advance_pc (4);
Syntax:	sub $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0010

SUBU – Subtract unsigned

Description:	Subtracts two registers and stores the result in a register
Operation:	$d ← $s - $t; advance_pc (4);
Syntax:	subu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0011

SW – Store word

Description:	The contents of $t is stored at the specified address.
Operation:	MEM[$s + offset] ← $t; advance_pc (4);
Syntax:	sw $t, offset($s)
Encoding:	1010 11ss ssst tttt iiii iiii iiii iiii

SYSCALL – System call

Description:	Generates a software interrupt.
Operation:	advance_pc (4);
Syntax:	syscall
Encoding:	0000 00-- ---- ---- ---- ---- --00 1100

XOR – Bitwise exclusive or

Description:	Exclusive ors two registers and stores the result in a register
Operation:	$d ← $s ^ $t; advance_pc (4);
Syntax:	xor $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d--- --10 0110

XORI – Bitwise exclusive or immediate

Description:	Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation:	$t ← $s ^ imm; advance_pc (4);
Syntax:	xori $t, $s, imm
Encoding:	0011 10ss ssst tttt iiii iiii iiii iiii