CONCORDIA UNIVERSITY
Department of Electrical and Computer Engineering
COEN 316 Computer Architecture
Lab 4 - CPU Datapath
Fall 2021

Introduction

The MIPS processor implemented in these labs contains 20 instructions divided into three categories: R (register) , I (Immediate) and J (Jump) instructions. Figure 1 gives the instruction format for each type of instruction.

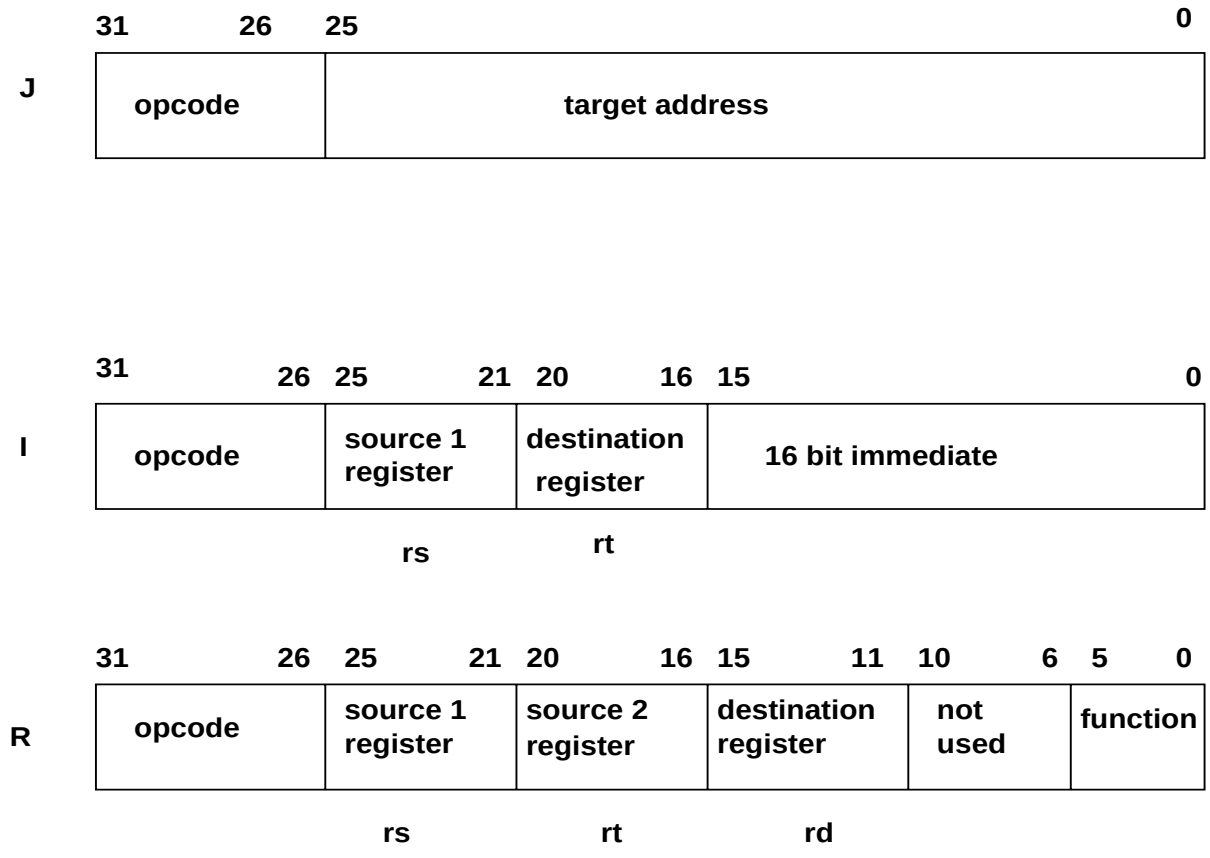

Figure 1: Instruction formats for R, I, and J type of instructions. [1]

Table 1 lists the 7 register type of instructions. The assembly language syntax of these instructions is of the form : *operation rd, rs, rt* meaning *rd = rs operation rt* with *rs* and *rt* specifying the two source registers (the 5 bit fields within the R-format instructions) and *rd* is the destination register. The *opcode* and *func* bits within the instruction are used to distinguish among the different instructions and will be the focus of Lab 5 (the control unit).

Table 1: Register instructions [2]

Instruction	Assembly Language	Meaning
Addition	`add rd, rs,rt`	rd = rs + rt
Subtraction	`sub rd, rs, rt`	rd = rs - rt
Set less than	`slt rd, rs, rt`	rd = 1 if (rs < rt) else rd = 0
AND (logical)	`and rd, rs, rt`	rd = rs AND rt
OR (logical)	`or rd, rs, rt`	rd = rs OR rt
XOR (logical)	`xor rd, rs, rt`	rd = rs XOR rt
NOR (logical)	`nor rd, rs, rt`	rd = rs NOR rt

The register instructions execute as follows [1]:

Step 1. Access the register file to read out the two source registers *rs* and *rt*

Step 2. Make these register operands available to the ALU unit and tell the ALU which operation to perform.

Step 3. Write the ALU output to the specified destination register *rd*.

There are a total of 11 Immediate type instructions grouped into ALU operations involving immediate data as one of the source operands, memory access instructions (load/store word) and conditional branch instructions. Table 2 lists these I-format instructions.

Table 2: Immediate instructions [2]

Instruction	Assembly Language	Meaning
Load upper immediate	`lui rt, immediate`	rt = immediate_data::0x0000 (16 bits immediate concatenated with 16 0s as the least significant bits)
Set less than immediate	`slti rt,rs,immediate`	rt = 1 if (rs < immediate_data) else rt = 0
Add immediate	`addi rt,rs,immediate`	rt = rs + immediate_data
AND immediate	`andi rt,rs,immediate`	rt = rs AND immediate_data

Table 2: Immediate instructions [2]

Instruction	Assembly Language	Meaning
OR immediate	`ori rt, rs,immediate`	rt = rs OR immediate_data
XOR immediate	`xori rt,rs,immediate`	rt = rs XOR immediate_data
Load word	`lw rt, immediate(rs)`	rt = mem[rs + immediate]
Store word	`sw rt, immediate(rs)`	mem[rs + immediate] = rt
Branch less than 0	`bltz rs, there`	if (rs <0) goto there
Branch equal	`beq rs, rt, there`	if (rs = rt) goto there
Branch not equal to 0	`bne rs, rt, there`	if (rs /= rt) goto there

The immediate arithmetic instructions and the immediate logical instructions have a similar executions sequence to the register type of instructions except that register *rs* is fetched from the register file as one of the input operands to the ALU with the other ALU operand coming from the 16 bit immediate data field of the instruction (appropriately sign extended) and the ALU output is written to register *rt* (rather than register *rd*).

The two instructions which access main memory (Load word and Store word) involve [1] :

Step 1. Read register *rs* from the register file and use it as one of the ALU input operands

Step 2. Read out the 16 bit immediate data within the instruction and use it as the second ALU input operand (sign extended as necessary).

Step 3. Add the two operands to form the memory address.

Step 4. Either read from this address and store the data read out from memory into *rt* (for a `load` instruction) or store the contents of *rt* into the memory address (in the case of a `store` word)

There are two unconditional jump instructions (jump and jump register) as indicated in Table 3.

Table 3: Unconditional jump instructions [2]

Instruction	Assembly Language	Meaning
Jump	`j there`	goto there
Jump register	`jr rs`	goto address contained in rs

Datapath design

Figure 2 gives a **partial** datapath of the CPU. It contains the components which have already been designed in previous labs (the ALU, the register file, and the next-address unit) as well as additional hardware needed to complete the datapath. In what follows, a detailed explanation of the datapath is provided You are to **complete the design** of the datapath based upon this textual description.

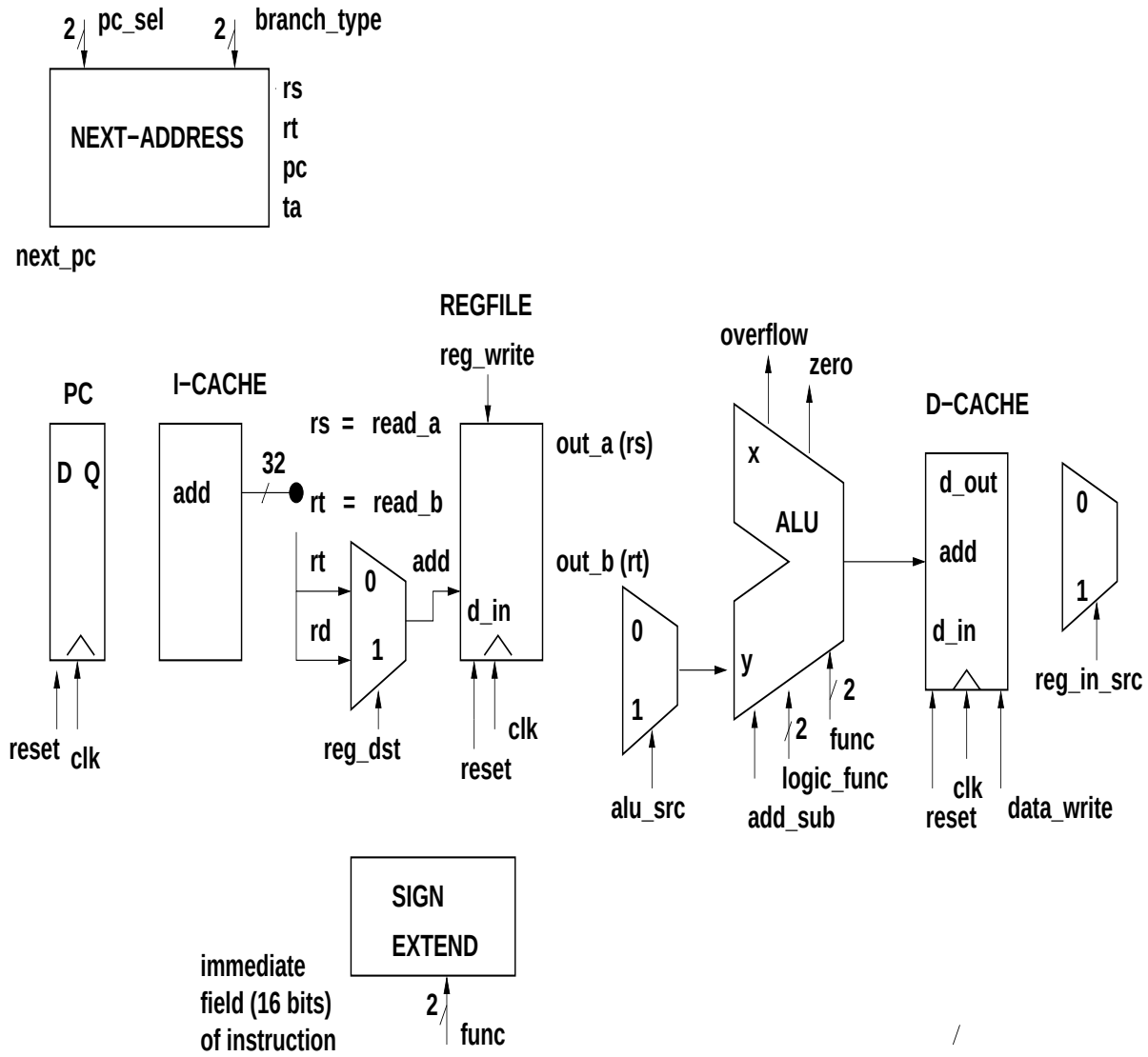

Figure 2: CPU datapath. [3]

PC register:

The program counter register is a 32-bit wide register with an asynchronous reset. The input to the PC register is the output of the next-address unit. The output of the PC register (the low-order 5 bits in order to reduce the size of the instruction cache memory) is used as the address input to the

instruction cache memory. The (full 32 bit) output of the PC is also an input to the next-address unit. The PC register may either be implemented as a VHDL process, or as a component via a port map instantiation.

Instruction Cache:

The I-cache unit stores the program machine code. It is a 32 location **read-only memory** (ROM) consisting of a 5-bit wide address input and a 32-bit wide data output (consisting of the 32 bits of machine code for the instruction stored at the addressed location). It's contents may be "hard-wired" to contain a program in machine code during the development and debugging stages of this lab. It is suggested that a VHDL process together with a case statement be used to implement the I-cache. Different programs may then be tested by commenting out the "old" cases with the new ones. The exact values for the opcode and func fields within an instruction is not critical in this lab (but they will be for the Lab 5), what is essential is value of the rs, rt, rd and immediate fields as these are required to test the functionality of the datapath. For example, consider the following assembly language program:

address memory contents

```
00000         addi r1, r0, 1 ; r1 = r0 + 1  = 0 + 1
00001         addi r2, r0, 2 ; r2 = r0 + 2  = 0 + 2
00010  there: add r2, r2, r1 ; r2 = r2 + r1 = r2 + 1
00011         j there        ; goto label there
```

(Note that in the MIPS processor register R0 is hardwired to the constant value of 0. We will implement this in our version by simply resetting the register value to all 0s upon assertion of the asynchronous reset (at system startup) and never writing to register R0.)

The above program would be stored in the I-cache ROM as follows:

```
          op   rs   rt   rd
        |    |    |    |    |
        |    |    |    |    |     |  fn       op   rd, rs, rt

00000  00100000000000010000000000000001  -- addi r1, r0, 1
00001  00100000000000010000000000000010  -- addi r2, r0, 2
00010  00000000010000010001000000100000  -- add  r2, r2, r1
00011  00001000000000000000000000000010  -- jump 00010
00100  00000000000000000000000000000000  -- don't care
```

The output of the I-cache is used to provide the read_a and read_b addresses of the register file (the rs field of the instruction is used as the read_a address and the rt field of the instruction is used as the read_b register file address input. The multiplexer at the write_address input of the register file (indicated with label "add" in Figure 2) allows for the selection of either the rt field or the rd field as the 5 bit address specifying which register is to be written into. A control signal (reg_dst) to be generated by the control unit determines this selection.

Register File:

The register file was previously designed in Lab 2 and it may be instantiated as a VHDL component with an appropriate VHDL port map statement.

ALU:

The arithmetic-logic unit was designed in Lab 1 and it may be similarly implemented as a VHDL component with a port-map statement.

The astute reader will note that there is a multiplexer which determines the second input (the 'y' input) to the ALU selecting between the out_b output of the register file or the output of the sign extend block . The ALU mux is controlled by a control-unit generated signal (alu_src). The sign extend unit performs the necessary sign extension of the 16 bit immediate data.

Sign Extension:

Some of the MIPS instructions require that the 16-bit immediate field (stored in bits 0 to 15) of I-format instructions be sign extended to a full 32-bit width. The exact manner of this sign extension depends upon the type of instruction to be executed. Table 4 provides a summary of the possible types of sign extension. In this table, the notation $i_{15}i_{14}i_{13}..i_1i_0$ refers to the 16 bits of immediate data stored within the instruction (the low-order 16 bits). The func is a 2-bit control signal (generated by the control unit) specifying the type of sign extension to be performed based upon the instruction.

Table 4: Sign extension formats

func	instruction type	sign extension	comments
00	load upper immediate	$i_{15}i_{14}i_{13}..i_1i_0$ 000..00	pad with 16 0s at least significant postions
01	set less immediate	$i_{15}i_{15}...$ $i_{15}i_{14}i_{13}..i_1i_0$	arithmetic sign extend (pad high order with copy of immediate sign bit i_{15})

Table 4: Sign extension formats

func	instruction type	sign extension	comments
10	arithmetic	$i_{15}i_{15}...\ i_{15}i_{14}i_{13}..i_1i_0$	arithmetic sign extend (pad high order with copy of immediate sign bit i_{15})
11	logical	$00...00\ i_{15}i_{14}i_{13}..i_1i_0$	high order 16 bits padded with 0s

Data cache:

The data cache acts as a small RAM -type of memory. The low order 5 bits of the ALU output are used to address one of 32 locations (each location is 32 bits wide). The data cache has an asynchronous reset, a clock (the writes into the data cache are synchronous with the clock), a data_write control signal (which acts as a enable, in other words, data is written into the memory at the next rising clock edge if data_write = '1'), and a 32 bit wide data output port. Since only the load/store instructions access the data cache, the out_b of the register file (which is the rt register) is connected to the d_in port of the data cache as per Figure 2. The data cache may be considered to be a single port register file and as such a VHDL clocked process may be used to implement it.

Associated with the data cache is a multiplexer which selects either the output of the data cache (in the case of a load instruction) or the ALU output (for other instructions such as the R-type of instructions in which the ALU output is to be written into the destination register). The output of this multiplexer provides input to the d_in of the register file.

Procedure

Complete the design of the datapath and then implement your design using VHDL. At this point, it is up to you whether you wish to design the datapath as a separate component (which will be used later in Lab 5 together with an instance of a control unit component) or whether you would like to add to the datapath designed in this lab additional VHDL code for the control unit as a separate process. The approach chosen will determine the VHDL entity for this lab , so no specific VHDL entity is given for this lab.

Requirements

As the minimal requirements, you are to provide:

1. A block diagram of your complete datapath showing the interconnections among all the various components within the datapath.

2. Modelsim simulation results for the Instruction cache, data cache and sign extend units developed in this lab. The objective is to show that these components have been thoroughly simulated to ensure correct functionality. It is also **strongly suggested** (but not required) to run the Vivado synthesis and implementation processes for the Instruction cache, data cache, and sign extend units to ensure that proper , synthesizeable VHDL coding constructs were used in their design. One need not constrain any top-level ports (no .xdc file required) and it is not necessary to run Bitgen to generate the .bit file. The intent is to verify that there are no errors during the Vivado synthesis and implementation phases .

3. The VHDL code for the complete datapath. Although not strictly required, it is highly suggested that you simulate the datapath prior to Lab 5 (to ensure that it is correct). The various control signals within the datapath may be set to specific values (within a DO file) so that the complete datapath may be simulated without having an actual control unit. The I-cache may be written to test several different instructions, or even several different small programs. This will be useful to have for Lab 5.

References

1. *Computer Architecture, From Microprocessors to Supercomputer*, Behrooz Parhami, Oxford University Press, ISBN 0-19-515455-x, 2006, p.245

2. *Computer Architecture, From Microprocessors to Supercomputer*, Behrooz Parhami, Oxford University Press, ISBN 0-19-515455-x, 2006, p.244.

3. *Computer Architecture, From Microprocessors to Supercomputer*, Behrooz Parhami, Oxford University Press, ISBN 0-19-515455-x, 2006, p.248.

Ted Obuchowicz
November. 3, 2016 - *"Nothing lasts forever, not even cold November rain."*

Revised: October 27, 2017. Modified to allow for greater design opportunity.

Revised: October 28, 2021. Modified for Vivado and Nexys A7 board.