ECE 443

MIPS-16, Part 3: Designing a 16-bit RISC style microprocessor

Lab #6: Instruction and data memory; basic components and datapath for I- and J- format instructions

Assigned **March 21, 2023**
Report due **April 4, 2023**

Introduction

The goal of this laboratory is to continue the design of a simple microprocessor. The microprocessor is a 16-bit machine that is based on the 32-bit Reduced Instruction Set Computer (RISC) machine that is presented in our textbook (Figure 4.11, also available on Blackboard).

Equipment:

The Altera Quartus II

Procedure:

1. Carry out attached worksheet and exercises.

MIPS-16:

The processor is a 16-bit machine with the following instruction formats;

R-Type opcode = I[15:12], Rs = I[11:9], Rt = I[8:6], Rd = I[5:3], funct = I[2:0]
I-Type opcode = I[15:12], Rs = I[11:9], Rt = I[8:6], imm = I[5:0]
J-Type opcode = I[15:12], adr = I[11:0]

The instruction set is given in Table 1.

Worksheet:

1. Instruction Fetch and PC Increment
   (a) Review Figure 4.6 in Patterson and Hennessey. Design a machine that adds **two** to a 16-bit number, **INCTWO**, in VHDL (*NOTE: look under 3. – the same ADD16 component, created out of ADD4, will be used for both steps).*
   (b) Design an instruction memory (**INS_MEM**) in VHDL, with one 16-bit data output (DOUT) and one 16-bit address input (ADDR). For simplicity, set up the address decoder (DCD) so that there are only 32 memory **words** and the memory is byte addressable. You can, however, assume that the words are aligned in memory at even addresses and that we will only read words (not bytes). The byte addresses should extend from 0x0000 to 0x003F. To implement this you will need to design a 5x32 decoder (**DCD5X32**), using an appropriate number of DCD3x8 components.

   You can implement the actual memory as an array of 16-bit STD_LOGIC_VECTOR signals. Set up the VHDL so that the instruction sequence (or program) is hardwired into the VHDL. You can simply assign a machine language representation of an instruction to a 16-bit signal (e.g.,

mem(0) <= "1000101110101101". For now, put in a simple program that does the following;

- loads some data from the lowest memory address into R1,
- loads some data from the next address into R2,
- adds R1 and R2 and stores the result into R3,
- stores the contents of R3 in the next memory address, and
- uses an infinite loop as a halt.

Use the fact that the contents of R0 will be hardwired to 0. Your solution should check to make sure the address sent to this component is within the allowed range of addresses.

2. Memory Access
   (a) Review Figure 4.10 in Patterson and Hennessey. Design a sign extend machine in VHDL that sign extends a 6-bit number to a 16-bit number, **SGNEXT6x16** (because our immediate field is 6 bits long).
   (b) Design a data memory (**DAT_MEM**) in VHDL, with two 16-bit inputs ADDR and DIN, two 1-bit inputs WE and RE (Write Enable and Read Enable), a CLK, and one 16-bit output, DOUT. For economy, set up the address decoder so that there are only 16 memory words and the memory is byte addressable. The byte addresses should extend from 0x0100 to 0x011F. Design a 4×16 decoder (**DCD4X16**) using two 3×8 decoders we created before. Design the data memory as a synchronous machine with a CLK input (remember, it's based on REG8).

   NOTE: *DATA MEMORY HAS TO BE* ***BASED ON THE PREVIOUSLY DESIGNED REG8 COMPONENT****. The output from the memory should be "high impedance" (Z) when RE is inactive. Use the enable signal of the register we previously designed.*

3. Branch instructions
   (a) Review Figure 4.9 of Patterson and Hennessey. Design a combinational "*shift left by one bit*" machine, **SHLONE**, which takes a 16-bit number and shifts it to the left, in VHDL.
   (b) Design a 16-bit adding machine, **ADD16** (using ADD4 that you already designed), in VHDL.

4. Jump Instructions
   (a) Design a 3-input, 16-bit multiplexer **MUX3x16** in VHDL.

Requirements:

1. **Follow the template provided on Blackboard**
2. The report should include all (new and changed) design files and a brief explanation for each of the components.
3. **The report should Include all relevant *.vhd files.**

4. The report should include annotated simulation files verifying that the components work as designed (i.e. read instruction, or store the data to a memory location and read it from there, etc.).
5. **One of the simulation files should show the process of reading the third instruction in your program.**
6. Record and tabulate the timing delays and chip resources used for INS_MEM and DATA_MEM components.
7. The report should include the table showing the binary code (machine language) of the program.

TABLE 1. MIPS-16 INSTRUCTION SET

Instruction	*HDL*	*Type*
add Rd, Rs, Rt	Rd := Rs + Rt	R-type
sub Rd, Rs, Rt	Rd := Rs - Rt	R-type
and Rd, Rs, Rt	Rd := Rt and Rd	R-type
or Rd, Rs, Rt	Rd := Rt or Rd	R-type

slt Rd, Rs, Rt	if (Rs < Rt) then
	Rd = 0001_{16}
	else
	Rd := 0000_{16}
		R-type

lw Rd, imm(Rs)	Rd := M(Rs + imm)	I-type
sw Rd, imm(Rs)	M(Rs + imm) := Rd	I-type
beq Rs, Rt, imm	if(Rs = Rd) then PC := PC + (imm×2)	I-type
j adr	PC := PC[15:13]:adr:0	J-type

TABLE 2. MIPS-16 ALU OPERATIONS

Operation	*SEL2*	*SEL[1:0]*
AND	0	00
OR	0	01
ADD	0	10
SUB	1	10
SLT	1	11

TABLE 3. MIPS-16 Instruction Set and Formats

Format	Fields	HDL	Example
R-Format	opcode:Rs:Rt:Rd:funct	Rd ← f(Rs, Rt)	`add $1, $2, $3`
I-format	opcode:Rs:Rt:imm	Rt ← M(Rs + imm)	`lw $1,addr($2)`
		if (Rt == Rs) then PC ← PC + (imm×2)	`beq $1,$2, addr`
J-format	opcode:addr	PC ← PC(15..p):addr:'0'	`j addr`

Note: The current value of the PC is the address of the next sequential instruction to be executed.

TABLE 4. MIPS-16 Instruction Opcodes and Function Fields			
Instruction	***Format***	***Hex Opcode***	***Function Field***
`add`	R	0_{16}	010
`sub`	R	0_{16}	110
`and`	R	0_{16}	000
`or`	R	0_{16}	001
`slt`	R	0_{16}	111
`lw`	I	B_{16}	n/a
`sw`	I	F_{16}	n/a
`beq`	I	4_{16}	n/a
`j`	J	2_{16}	n/a

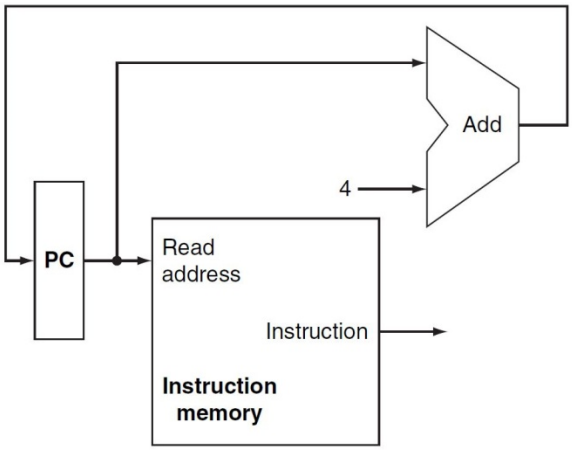

FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

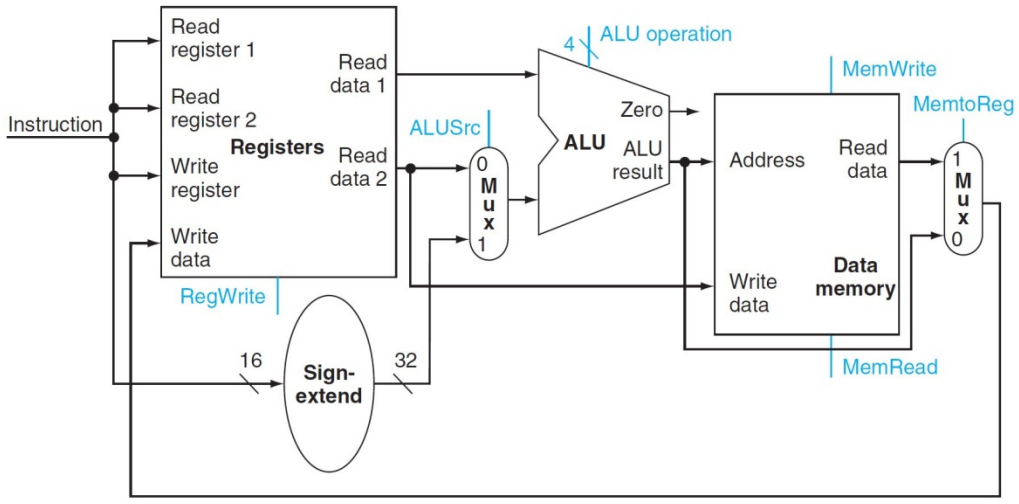

FIGURE 4.10 The datapath for the memory instructions and the R-type instructions.

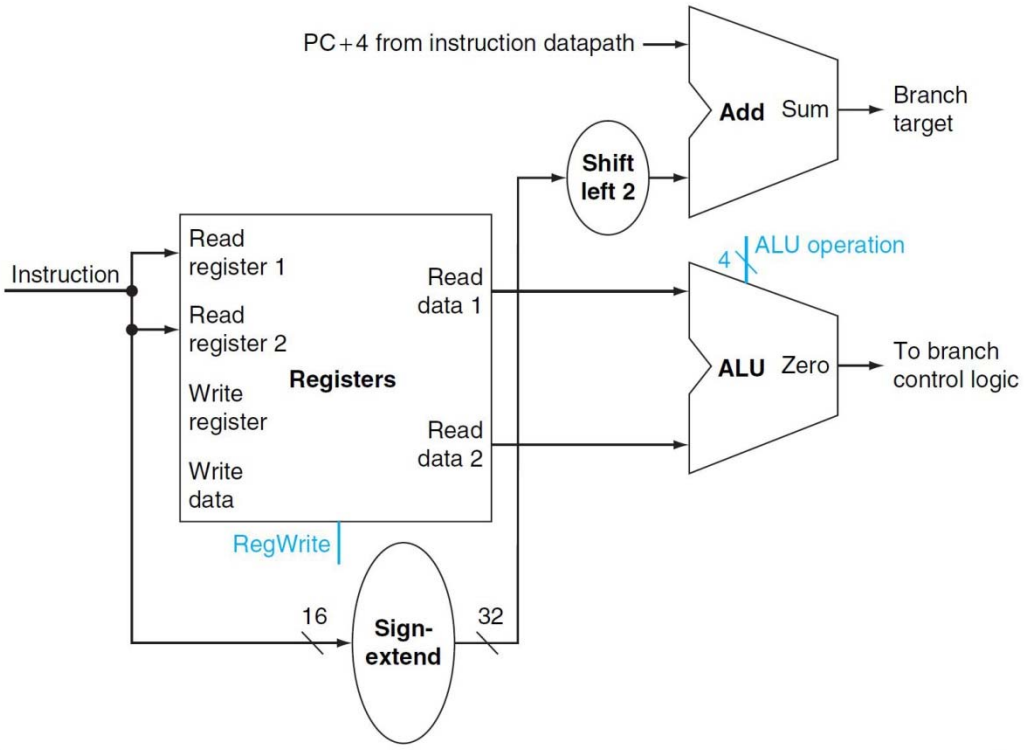

FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the "shift" is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only "sign bits." Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

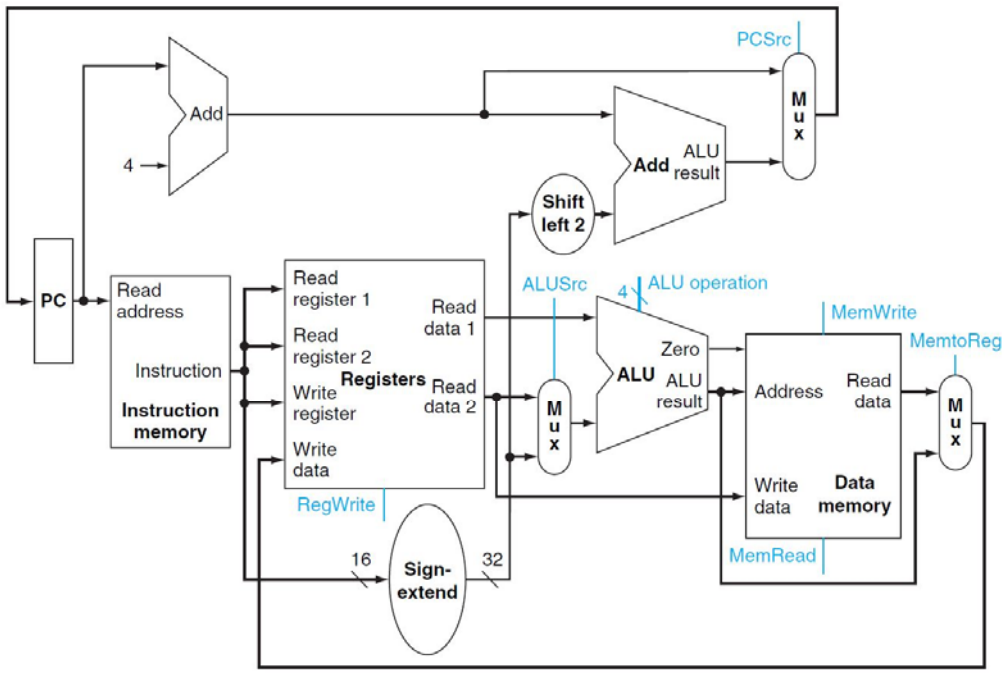

FIGURE 4.11 The simple datapath for the MIPS architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. An additional multiplexor is needed to integrate branches. The support for jumps will be added later.