ECE 0142

Summer 2015

Project Final Report

Instructor: Dr. Irvin Jones
Section: Summer 2015

Prepared By: Reshef Elisha

Report Grade: ___________

Table of Contents

Table of Figures

Table of Tables

Executive Summary

This project aims to design and implement a processor which can execute the ISA given in the assignment paper. From the ISA, a control scheme was designed, and an architecture was built around that control scheme. The final implementation was then done in Python, to emulate how a processor with the designed architecture would operate. Once the processor was implemented, it had to read in binary instruction input. To make writing that input easier an assembler was written, also in Python, which would generate binary data from assembly instruction files. Once the binary instructions are generated, they are executed in the main loop of the processor script.

1. Introduction

This project aims to implement a processor that will work with the ISA described in Section 2. The schematic for the processor, from which the implementation and code were designed, can be found in Figure 1 in 3. From there, the code was written, and is described in 4, and can be found in Appendix A: Processor code. An assembler was written, for which the code can be found in Appendix B: Assembler Code.

2. Instruction Set Architecture (ISA): System Requirements

The ISA for the processor includes 9 instructions, divided into 3 instruction types. I-type, which deals with immediate values and memory related , R-type, which deals with registers and their values, and J-type, used exclusively for the Jump instruction. The instruction types are distinguished by the first two most significant bits (15..14) of the opcode, in a Big-endian formatting, which I named the Group code. The following two bits of the 4-bit opcode designate which function the ALU performs, and are also used to denote individual instructions inside groups.

2.1 I-type instructions

The I-type instruction group has the group code of [00], and contains the instructions **addi** and **beq**, and **NOP**. It has the structure described in Table 1.

2 Bits (15..14)	2 Bits (13..12)	4 Bits (11..8)	8 Bits (7..0)
00	FUNCTION CODE	REGISTER A	IMMEDIATE

Table 1: I-type instruction format

2.1.1 ADDI

Format: **addi RA, IMM**

The **addi** instruction is used to add a register's value to an immediate value. It has the function code [00] (add) to make the full opcode [0000]. The RTL notation for this instruction is:

RA $\leftarrow$ RA + Sign Ext[Imm].

2.1.2 BEQ

Format: **beq RA, IMM**

The **beq** instruction is used to jump the program counter's value relative to itself. It has the function code [01] (sub) to make the full opcode [0001]. The RTL notation for this instruction is:

(RA == 0) : PC + Sign Ext[Imm].

You will notice that the function code for this instruction is subtract, but that does not affect the final outcome since the ALU result is not needed in a **beq** instruction.

2.1.3 NOP

Format: **NOP**

The **NOP** instruction is used to skip steps if a wait is needed. If this were a pipelined architecture, it would also be used to wait in case of a data or control hazard. Its function code is [00] (add) to make the full opcode [0000]. This instruction is technically an **addi** operation, but since all arguments are 0, all it does is add 0 to register 0, thereby changing nothing.

2.2 I-type instructions that deal with memory

The second class of I-type instruction group has the group code of [10]. It handles the three memory related instructions: **la**, **lw**, **sw**. The instruction format for la is specified in Table 2.

2 Bits (15..14)	2 Bits (13..12)	4 Bits (11..8)	8 Bits (7..0)
10	00	REGISTER A	IMMEDIATE

Table 2: Second class I-type instruction format

You'll notice that in **sw** and **lw** don't use the immediate value, as such it is left as 0 for those 2 instructions.

2.2.1 LA

Format: **la RA, IMM**

The **la** instruction is used to set the memory access register (MAR). This instruction sets the MAR to RA + immediate (shift) value. It has the function code [00] (add) to make the full opcode [1000]. The RTL notation for this instruction is:

MAR $\leftarrow$ RA + Sign Ext[Imm].

2.2.2 LW

Format: **lw RA**

The **lw** instruction is used to load a word from memory. It uses the MAR to pick which address to load from, and loads it to register A. It has the function code [01] (sub), to make the full opcode [1001]. The RTL notation for this instruction is:

RA $\leftarrow$ DataMem[MAR].

2.2.3 SW

Format: **sw RA**

The **sw** instruction is used to store a word to memory. It uses the MAR to pick which address to store to, and stores register A's value to it. It has the function code [10] (nor), to make the full opcode [1010]. The RTL notation for this instruction is:

DataMem[MAR] $\leftarrow$ RA

2.3 R-type instructions

The R-type instruction group has the group code of [01], and contains the instructions **add** and **sub**, and **nor**. It has the structure described in Table 3.

2 Bits (15..14)	2 Bits (13..12)	4 Bits (11..8)	4 Bits (7..4)	4 Bits(4..0)
01	FUNCTION CODE	REGISTER A	REGISTER B	Not used

Table 3: R-type instruction format

2.3.1 ADD

Format: **add RA, RB**

The **add** instruction is used to add a register's value to another register's value. It has the function code [00] (add) to make the full opcode [0100]. The RTL notation for this instruction is:

RA 🡐 RA + RB.

2.3.2 SUB

Format: **sub RA, RB**

The **add** instruction is used to subtract a register's value from another register's value. It has the function code [01] (sub) to make the full opcode [0101]. The RTL notation for this instruction is:

RA 🡐 RA - RB.

2.3.3 NOR

Format: **nor RA, RB**

The **nor** instruction is used to nor two registers' values. It has the function code [10] (nor) to make the full opcode [0110]. The RTL notation for this instruction is:

RA 🡐 RA ↓ RB.

2.4 J-Type Instructions

The J-type instruction group has the group code of [11], and only contains the instructions **j**. It has the structure described in Table 4.

2 Bits (15..14)	2 Bits (13..12)	12 Bits (11..0)
11	00	Jump Value

Table 4: J-type instruction format

2.4.1 J

Format: **j JVALUE**

The **j** instruction is used to jump the PC to an immediate jump value. It has the function code [00] (add), to make the full opcode [1100]. The RTL notation for this instruction is:

PC $\leftarrow$ PC[15..12] + JumpValue.

2.5 Memory Addressing Modes

There are four main memory addressing modes, and all of them are doable with the given ISA and implementation.

2.5.1 Register Addressing

Register addressing implies using the value stored in a register to perform an operation, which can be done with any of the three R-type instructions.

add RA,RB

sub RA,RB

nor RA,RB

2.5.2 Base Displacement Addressing

Base Displacement Addressing means taking a register as base, then adding an immediate displacement to it, and accessing that address from memory. Base Displacement Addressing is possible with the given architecture using a combination of **la** and **lw**.

la RB,IMM

lw RA

2.5.3 Immediate Addressing

Immediate Addressing is the practice of using a register and an immediate value for an operation. It can be done with the two I-type instructions.

addi RA,IMM

beq RA,IMM

2.5.4 Pseudo-Direct Addressing

Pseudo Direct is used during jumps. It means to truncate the last 12 bits of the program counter and replace them with the give jump value. This can be done with the **j** instruction.

j JVAL

3. Processor Architecture

My processor architecture is based on a MIPS architecture that was modified to accommodate some of the different instructions given as part of the assignment. The most obvious change is the addition of the MAR (Memory Access Register) that is there to allow for the separation of memory and register access into different instructions.

3.1 Datapath

The datapath is similar to that of a MIPS processor. The instruction data is fetched from the instruction memory, then each part of the instruction is sent where it is needed. The opcode, bits 15 through 12, are sent directly and only to the control unit, which then generates the control signals. The control unit will be discussed further in Section 3.2. Bits 11 through 8 are sent to Register A input of the register file, and Bits 7 through 4 are sent to Register B. Registers 7 through 0 are sent into the sign extension unit, and bits 11 through 0 are directed to the jump multiplexer. The entire schematic can be seen in Figure 1.

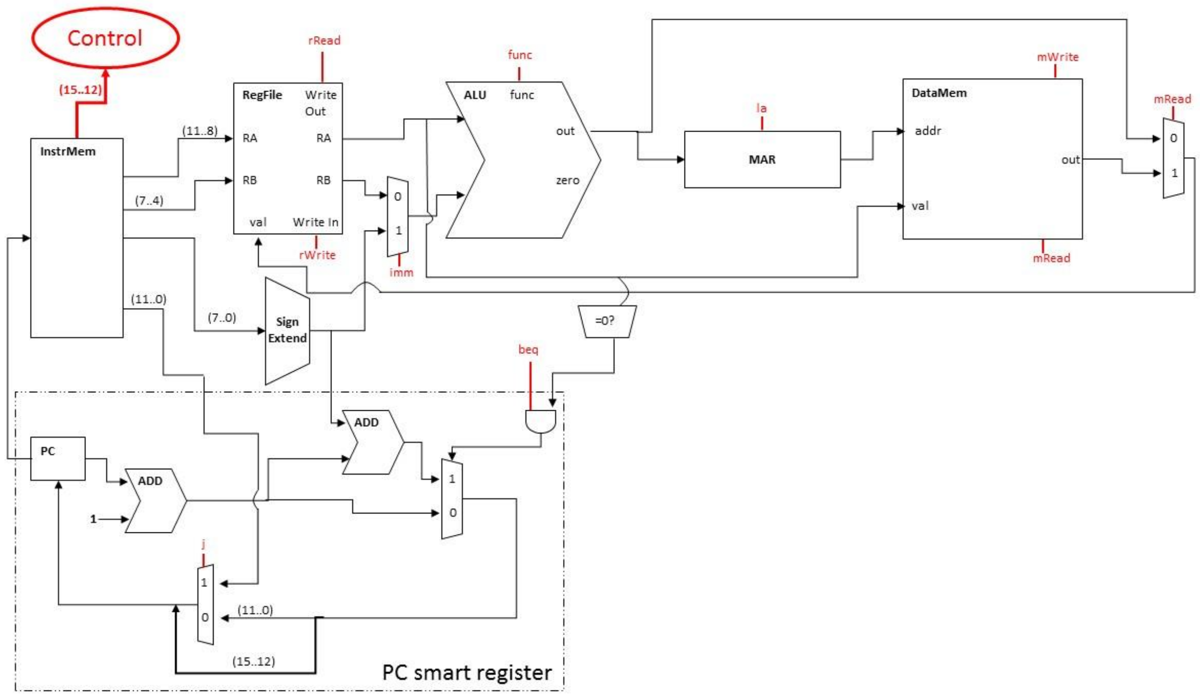

Figure 1: Architecture Schematic

3.1.1 Instruction Memory

The Instruction Memory is read one 16-bit word at a time. It takes in a value from the Program Counter and outputs the 16 bit word.

3.1.2 Register File

The Register File takes in two 4-bit values, RA and RB and outputs two 16-bit values, which are the values stored in registers RA and RB. It also takes in a write value, which will be written only if the ‘Write In’ control flag is set. It will also only output values if the ‘Write Out’ control flag is set, otherwise it will output 0.

3.1.3 ALU

The ALU takes in two values, A and B, and performs a function on them. The function is defined by the 2-bit function code sent from the control unit into the input ‘func’. The function codes are mapped as shown in Table 5. The ALU then outputs two values, ‘out’, which is the result of the operation, and a ‘zero’ 1-bit fag, which is set if the result is equal to 0.

00	add
01	sub
10	nor

Table 5: Function code definitions

3.1.4 MAR

The Memory Access Register is set by the command **la**. It takes in an input from the ALU, but only sets it if the ‘la’ control flag is set.

3.1.5 Data Memory

The Data Memory is read and written into one word at a time. The address is defined by the value of the MAR. If the control flag ‘mRead’ is set, the data memory outputs the value stored at the memory location defined by the address. If the control flag ‘mWrite’ is set, the value that is sent from the register file into the ‘val’ input will be stored in the memory location defined by the address.

3.1.6 Sign Extend

The sign extension unit takes in an 8-bit value and outputs a sign-extended 16-bit value that is generated by repeating the most significant bit of the 8-bit value 8 times, and sticking that in front of the value itself.

3.1.7 Program Counter system

The Program Counter system consists of all of the blocks in the lower half of Figure 1. First, the value 1 is added to the program counter. Unlike MIPS, which adds 4 to the program counter, we only add 1, as the instruction memory is read by the word. After that, if the ‘beq’ control flag is set and the ALU flag ‘zero’ is also set, the Program Counter value has the immediate value added to it (this happens only during the **beq** instruction). Lastly, if the ‘j’ control flag is set, the 12 least significant bits of the Program Counter value are discarded and replaced with the jump value set by bits 11 through 0 of the instruction. This only happens during a **j** instruction.

3.2 Controller

The Controller is the unit that generates all of the control flags referred to in Section 3.1. It takes in one 4-bit input which comes from the bits 15 through 12 of the current instruction, and using combinational logic outputs 8 1-bit flags and 1 2-bit function code. The control codes generated are described in Table 6. For each instruction, the opcode is given, and from it the flags ‘rRead’, ‘rWrite’, ‘imm’, ‘la’, ‘mRead’, ‘mWrite’, ‘beq’, and ‘j’ are generated. The ‘func’ column does not represent an actually generated flag, but only serves as a guide to describe what the last two bits of the opcode mean.

Function	Opcode		func	read	write	imm	la	mread	mwrite	beq	j
la	10	00	add	1	0	1	1	0	0	0	0
lw	10	01	sub	0	1	0	0	1	0	0	0
sw	10	10	nor	1	0	0	0	0	1	0	0
add	01	00	add	1	1	0	0	0	0	0	0
sub	01	01	sub	1	1	0	0	0	0	0	0
addi	00	00	add	1	1	1	0	0	0	0	0
nor	01	10	nor	1	1	0	0	0	0	0	0
beq	00	01	sub	1	0	1	0	0	0	1	0
j	11	00	add	0	0	0	0	0	0	0	1

Table 6: Control codes per instruction

4. Virtual Machine Simulator/Emulator

The processor's emulation was written in python, and coded to be as efficient as possible while staying accurate to the original schematic. The code can be found in Appendix A: Processor code.

4.1 Implementation Platform

The entire emulator is written in python, as it is a language that makes it very straightforward to go from binary string to integer numbers, and convert between the two. It was written in python version 2.7, which is the most common version of the language. The memories (Instruction and Data) were implemented as text files that can be read from and written to.

4.2 System Architecture

Each functional unit is written in python as a unique function. The control flags and the MAR are global variables. There are some additional helper functions such as a two's complement converter in order to help other functions such as the ALU stay short, and reduce the amount of repeated code. These helper functions don't emulate a physical component, but are merely there to help other functions be more accurate while keeping the entire program short.

4.3 Operating Instructions

Make sure you have python 2.7 installed on your system. To run the emulator, you will first want to translate your assembly instructions into proper binary and place them in the instruction memory. To do that, write your assembly instructions using the specified ISA into assm.txt, and run the command `python assembler.py` (the code for which can be found in Appendix B: Assembler Code). This will create or edit the file `instMem.txt` and place the binary instructions into it. If you do not want to write you own instructions you do not need to run this step, as a program has been prewritten into both `assm.txt` (assembly version) and `instMem.txt` (binary version). Finally, to run the simulation, run the command `python processor.py`.

5. Summary

This paper presented my implementation of a MIPS style processor, modified to fit the assigned ISA. It was written in python and can be run on any machine that has python 2.7.

6. Documentation

The documentation used to create this project, other than the assignment paper, is the Python 2.7 documentation which can be found at https://docs.python.org/2/ .

Appendix A: Processor code

```
#################################################################
#                                                               #
#processor.py                                                   #
#Created by: R.H. Elisha     Date: 07-29-2015                   #
#Inputs: instructions input to instMem.txt                      #
#Outputs: Processor result, data output to dMem.txt             #
#Description: This is a processor simulation written to emulate #
#   a processor I designed.                                     #
#Side Effects: dMem.txt may be modified                         #
#################################################################

import sys

#READING INSTRUCTION AND DATA MEMORY
try:
    instMm = open("instMem.txt", 'r')
except:
    sys.exit("No instruction Memory")
try:
    dMm = open("dMem.txt", 'r')
except:
    sys.exit("No data Memory")
instMem = list(instMm) #initializing instruction memory list
dMem = list(dMm)#initializing data memory list
dMem = [m.replace('\n','') for m in dMem] #removing all unnecessary line
breaks and empty lines from the data memory
instMm.close() #closing the file as we are done with it
dMm.close() #closing the file as we are done with it
pC = 0 #starting the program counter off at 0

#GLOBAL CONTROL VARIABLES
rRead = False
rWrite = False
imm = False
func = 0
la = False
mRead = False
mWrite = False
beq = False
j = False
registerFile = ['0000000000000000']*16
MAR = '0'*16
#END GLOBAL CONTROL

def control(opCode): #takes a 4 bit opcode, changes global flag variables and
function code
    global func, rRead, rWrite, imm, la, mRead, mWrite, beq, j, registerFile,
MAR
    func = int(opCode[2:4],2)
    if opCode[0:2] == '01': #r-type
        rRead,rWrite,imm,la,mRead,mWrite,beq,j =
True,True,False,False,False,False,False,False
    elif opCode[0:2] == '00': #i-type
        rRead,rWrite,imm,la,mRead,mWrite,beq,j = True,func ==
0,True,False,False,False,func==1,False
    elif opCode[0:2] == '10': #i-type section 2
        rRead,rWrite,imm,la,mRead,mWrite,beq,j =
opCode[3]=='0',opCode[3]=='1',func==0,func==0,opCode[3]=='1',func==2,False,Fa
lse
    elif opCode[0:2] == '11': #j-type
        rRead,rWrite,imm,la,mRead,mWrite,beq,j =
False,False,False,False,False,False,False,True

def regFileRead(rA, rB): #rA, rB are 4 bit binary strings
    return {'rA':registerFile[int(rA,2)],'rB':registerFile[int(rB,2)]}if
rRead else{'rA':'0'*16,'rB':'0'*16}
def regFileWrite(rA, val): #rA is a string, val is a string
```

```
    registerFile[int(rA,2)] = val if rWrite else registerFile[int(rA,2)]
def signExt(immd): #immd is an 8 bit binary string
    return immd[0]*8+immd
def twosComp(bitString): #bitstring is any length binary string
    return int(bitString,2) if bitString[0]=='0' else int(bitString,2) -
(2**len(bitString))
def toTwosComp(dec): #dec is a decimal int
    return format(2**16+dec, '008b') if dec<0 else format(2**16+dec,
'008b')[1:17]
def bitwiseNOR(bsA, bsB): #takes to bitStrings and does a bitwise NOR on them
    return ''.join('1' if bsA[x] == '0' and bsB[x] == '0' else '0' for x in
range(0,len(bsA)))
def ALU(rA,rB): #rA, rB are strings, returns a dictionary with string value
for 'out' and boolen 'zero'
    if func == 0: #function add
        result = twosComp(rA)+twosComp(rB) #result is int
        return {'out': toTwosComp(result), 'zero': result==0}
    elif func == 1: #function subtract
        result = twosComp(rA)-twosComp(rB)
        return {'out': toTwosComp(result), 'zero': result==0}
    elif func == 2: #function NOR
        result = bitwiseNOR(rA, rB)
        return {'out': result, 'zero': result=='0'*16}
def MEM(rA): #rA is a string
    if mRead:
        return {'out':dMem[int(MAR,2)]} #read from memory, returns a
dictionary
    elif mWrite:
        dMem[int(MAR,2)] = rA #write to memory
        return None

while pC < len(instMem): #main program loop, runs until the PC reaches the
end of the instructions
    line = instMem[pC][0:16].replace('\n','') #gets the instruction memory
line
    opCode = line[0:4] #opcode is 4 MSB
    rA = line[4:8] #Register A
    rB = line[8:12] #Register B
    immd = line[8:16] #immidiate value
    jV = line[4:16] #jump value
    control(opCode) #opCode is a 4bit string
    regs = regFileRead(rA, rB) #rA, rB are 4 bit strings
    A = ALU(regs['rA'],(signExt(immd) if imm else regs['rB'])) #ALU takes in
register A's value and either register B's value or the immediate
    MAR = A['out'] if la else MAR #MAR is set only if la flag is set
    dM = MEM(regs['rA']) #get dataMemory
    val = dM['out'] if mRead else A['out'] #pick which value to write to
registers (memory or ALU)
    regFileWrite(rA, val) #write to register file
    print "\n\nPC: "+str(pC) #print the PC
    print registerFile #show the registers
    if mWrite: #if a data memory value was modified
        print 'Modified Memory: value:' + dMem[int(MAR,2)] + ' at address: '
+ MAR #print it
    print '-----------------------------------------------------------------
--------------' #aesthetic separator
    pC += 1 #increment PC
    pC = pC + twosComp(signExt(immd)) if (beq and int(regs['rA'],2)==0) else
pC #if beq flag is set, add extra value to PC
    pC = int((format(pC, '016b')[0:4]+jV),2) if j else pC #if j flag is set,
set the PC to the jump value

dMm = open("dMem.txt",'w') #reopen the data memory file
for line in dMem: #write each line to it.
    print line #print written line at the end of the program
    dMm.write(line[0:16]+"\n") #write the word
dMm.close() #clsoe the file after writing
```

The code can also be found as text in the file `processor.py`

Appendix B: Assembler Code

```
import sys
#OPEN FILES NEEDED FOR READING AND WRITING
try:
    assm = list(open("assm.txt", 'r')) #read from assm.txt
    instMem = open("instMem.txt", 'w') #write to instMem.txt
except:
    sys.exit("Error in the file system somewhere") #Catch the error if it
happens and exit

def toTwosComp(dec): #dec is a decimal int
    return format(2**8+dec, '008b') if dec<0 else format(2**8+dec,
'008b')[1:9] #return two's complement string

for line in assm: #for each line in the assembly code
    l = line.partition(' ') #separate it at the first space
    comm = l[0] #command is before the space
    args = l[2].replace(' ','') #remove spaces from arguments
    args = l[2].replace('\n','') #remove newlines from arguments
    print comm
    if comm == 'addi': #addi instruction
        parts = args.partition(',')
        rA = format(int(parts[0]),'004b')
        rA = rA[len(rA)-4:len(rA)]
        imm = toTwosComp(int(parts[2]))
        inst = '0000'+rA+imm
    elif comm == 'beq': #beq instruction, etc...
        parts = args.partition(',')
        rA = format(int(parts[0]),'004b')
        rA = rA[len(rA)-4:len(rA)]
        imm = toTwosComp(int(parts[2]))
        inst = '0001'+rA+imm
    elif comm == 'add':
        parts = args.partition(',')
        rA = format(int(parts[0]),'004b')
        rA = rA[len(rA)-4:len(rA)]
        rB = format(int(parts[2]),'004b')
        rB = rB[len(rB)-4:len(rB)]
        inst = '0100'+rA+rB+'0000'
    elif comm == 'sub':
        parts = args.partition(',')
        rA = format(int(parts[0]),'004b')
        rA = rA[len(rA)-4:len(rA)]
        rB = format(int(parts[2]),'004b')
        rB = rB[len(rB)-4:len(rB)]
        inst = '0101'+rA+rB+'0000'
    elif comm == 'nor':
        parts = args.partition(',')
        rA = format(int(parts[0]),'004b')
        rA = rA[len(rA)-4:len(rA)]
        rB = format(int(parts[2]),'004b')
        rB = rB[len(rB)-4:len(rB)]
        inst = '0110'+rA+rB+'0000'
    elif comm == 'la':
        parts = args.partition(',')
        rA = format(int(parts[0]),'004b')
        rA = rA[len(rA)-4:len(rA)]
        imm = format(int(parts[2]),'008b')
        imm = imm[len(imm)-8:len(imm)]
        inst = '1000'+rA+imm
    elif comm == 'lw':
        rA = format(int(args),'004b')
        rA = rA[len(rA)-4:len(rA)]
        inst = '1001'+rA+('0'*8)
    elif comm == 'sw':
        rA = format(int(args),'004b')
        rA = rA[len(rA)-4:len(rA)]
        inst = '1010'+rA+('0'*8)
    elif comm == 'j':
        jVal = format(int(args),'012b')
        jVal = jVal[len(jVal)-12:len(jVal)]
        inst = '1100'+jVal
    else: #something else, just leave a NOP
        inst = '0'*16
    instMem.write(inst+'\n') #write the line to instruction memory

print 'DONE' #print DONE if everything went well
```

The code can also be found in assembler.py

References

Dr. Irvin Jones

Joydeep Rakshit

Python documentation