German University in Cairo
CSEN601 – Computer System Architecture
Project Report – Mips Simulator
Submitted by:
Mohamed Ashraf Heikal – 22-0505
Nourhan Mohamed AbdelTawab – 22-1815
Nourhan Zakaria Khalafallah – 22-3385
T08

A Brief Description of the project:

The program takes the assembly file path as an argument and then reads every instruction into the instruction memory which is then read by the fetch function and passed to the decode. The program ends when the last instruction in the instruction memory is executed.

We have dictionaries that translate the text representation of the instructions and registers to their binary equivalent. for use within the program and during printing

The registers all start out as 0 except the $sp which starts out as the last address in the memory.

We have a control unit that gets the opcode and returns a dictionary of all the control signals.

We have functions that execute r, j and i format instructions separately based on the opcode of the function. Once the execute function has finished it calls the memory with the required action read/write.

We have a function that implements the memory write or read and write back. It also handles reading and writing of half words and bytes and stores them in memory accordingly. It also handles the printing of the used part of the memory. The maximum memory size is 4 GB since that is the largest address the instruction format can support. However from the point of view of the simulator the memory is allocated when it is first written to so as to avoid the allocation of 4GBs every time the program is run even though they may not be needed.

The decode function takes the entire text instruction and then uses the different helper methods to find the instruction type and then call the appropriate execute function according to instruction type.

All the separate methods handle the printing of the wires coming in and out of them when called

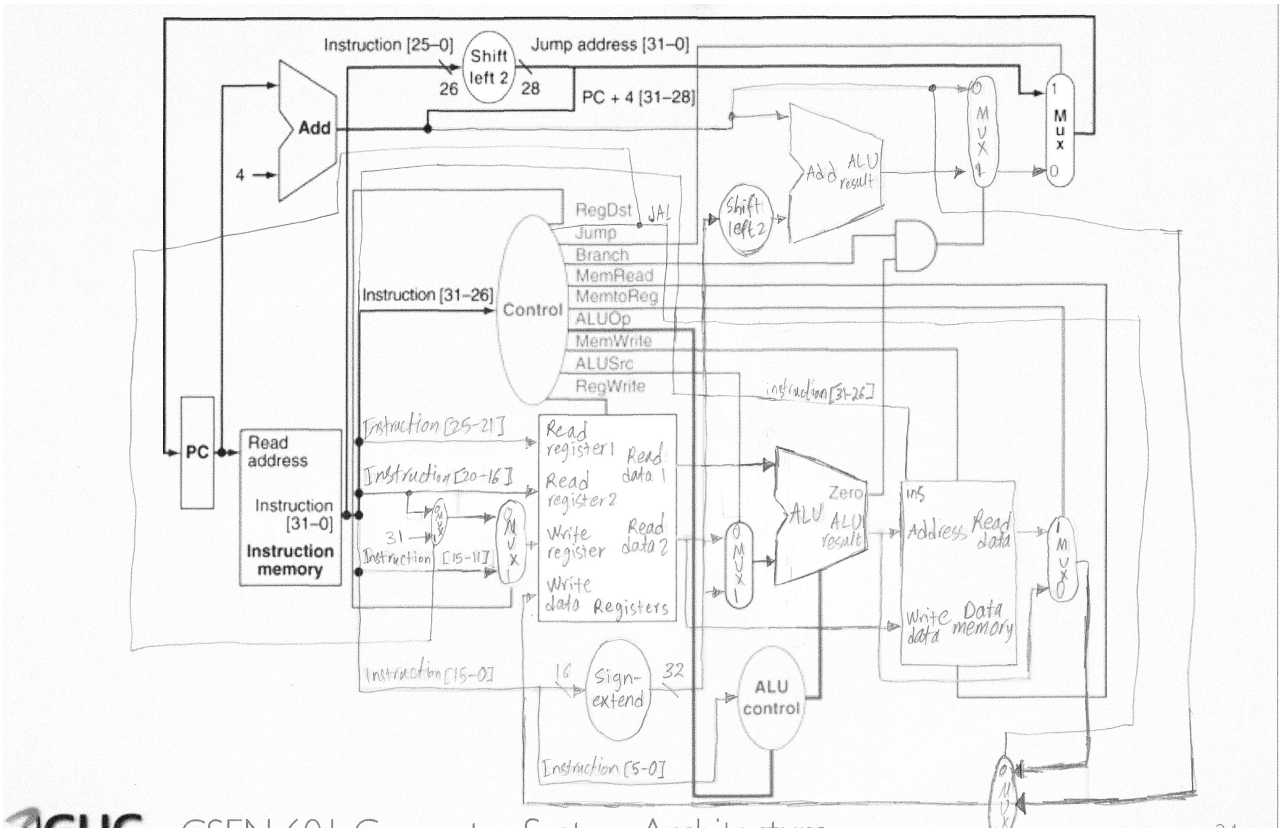
The Control Unit:

This is the simple pseudo code for the control unit

```
reg_dst = branch = mem_read = mem_to_reg = mem_write = reg_write = alu_src = jump = False
alu_op = 0
if operation in r_instructions:
  reg_dst = True

if operation == "beq" or operation == "bne":
  branch = True

if operation == "jal" or operation == "j" or operation == "jr":
  jump = True

if operation == "lw" or operation == "lh" or operation == "lb" or operation == "lhu" or operation ==
"lbu":
  mem_read = True
  mem_to_reg = True

if operation == "sw" or operation == "sh" or operation == "sb":
  mem_write = True

if not branch or not jump or not mem_write:
  reg_write = True

if not branch or not jump or not reg_dst:
  alu_src = True
```

Data Path:

Here is a snapshot of the implemented datapath:



Assumptions:

We assumed that there are no labels and branch instructions actually enter the number of instruction to skip or go back to and j instruction takes a 0-based address. Also we assumed that the entire file would be executed and the execution would halt when the last instruction has finished. All instructions and registers are assumed to be lowercase and we also assumed that users will not enter invalid syntax into the files to be executed.

Three sample programs:

program 1:

```
add $t0, $zero, $zero
addi $t1, $zero, 5
beq $t1, $t0, 3
sub $t2, $t2, $t1
srl $t1, $t1, 1
j 2
slt $t3, $t2, $t1
```

program 2:

```
jal 1
j 3
jr $ra
add $zero, $zero, $zero
```

program 3:

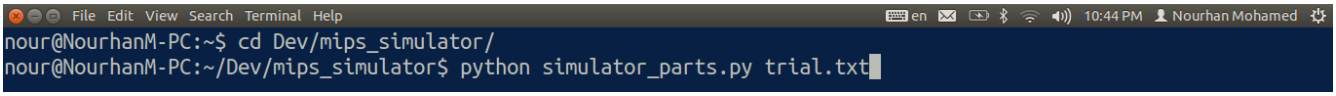
```
andi $t3, $t1, -1
ori $t3, $t3, 5
sll $t3, $t3, 2
ori $t3, $t3, 0
beq $t4, $t5, 2
lw $t4, 0($t3)
lh $t4, 0($t3)
lb $t4, 0($t3)
lhu $t4, 0($t3)
lbu $t4, 0($t3)
sw $t4, 0($t3)
sh $t4, 0($t3)
sb $t4, 0($t3)
lui $t4, 0($t3)
and $t4, $t4, $t4
or $t4, $t4, $t4
nor $t4, $t4, $t4
```

How the work was split across members:

Nourhan Mohamed: Took the fetch and decode functions and the main method
Nourhan Zakaria: Took the memory, read and write back functions and related helper methods
Mohamed Ashraf: Took the execute function and the part that parses the file into the instruction memory

Sample Run Guide:

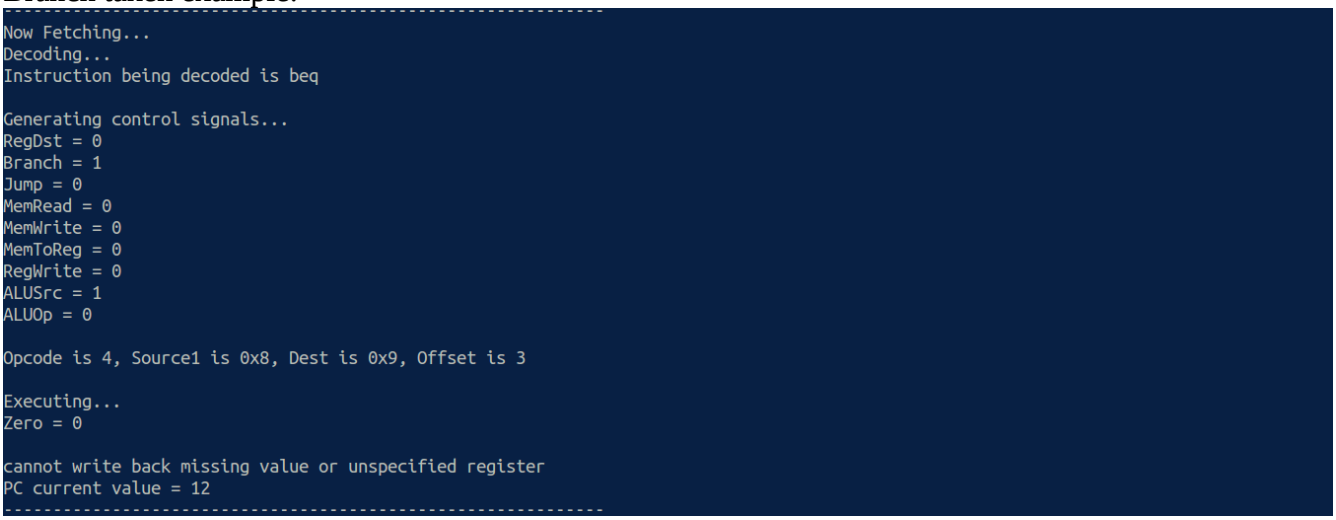
To actually run the simulation, you need python 2.7.3 installed and running on the maching.
On a terminal, type: python simulator_parts.py <file_path>
where file_path is the path of the file containing the mips code as shown below



During the simulation, Each clock cycle is printed to the user showing the instruction being fetched in text, followed by the generated control signals in the decode phase, then the fields of the instruction according to its type. For example, the opcode, rs, rt, rd, shamt and function are printed and so on. Then The memory and register files are printed if altered. The current PC value is also printed. Illustrations are shown below via snapshots

Jump example:

Branch taken example:



Add example:

```
------------------------------------------------------------
Now Fetching...
Decoding...
Instruction being decoded is add

Generating control signals...
RegDst = 1
Branch = 0
Jump = 0
MemRead = 0
MemWrite = 0
MemToReg = 0
RegWrite = 1
ALUSrc = 1
ALUOp = 0

Opcode is 0, Function is 32, Source1 is 0x0, Source2 is 0x0, Dest is 0x8, Shamt is 0

Executing...
Zero = 0

Executing write back stage ...
```

Addi example:

```
------------------------------------------------------------
Now Fetching...
Decoding...
Instruction being decoded is addi

Generating control signals...
RegDst = 0
Branch = 0
Jump = 0
MemRead = 0
MemWrite = 0
MemToReg = 0
RegWrite = 1
ALUSrc = 0
ALUOp = 0

Opcode is 8, Source1 is 0x0, Dest is 0x9, Offset is 5

Executing...
Zero = 0

Executing write back stage ...
```

Reg file print:

```
contents of register file:
reg $zero: 0x0
reg $at: 0x0
reg $v0: 0x0
reg $v1: 0x0
reg $a0: 0x0
reg $a1: 0x0
reg $a2: 0x0
reg $a3: 0x0
reg $t0: 0x0
reg $t1: 0x5
reg $t2: 0x0
reg $t3: 0x0
reg $t4: 0x0
reg $t5: 0x0
reg $t6: 0x0
reg $t7: 0x0
reg $s0: 0x0
reg $s1: 0x0
reg $s2: 0x0
reg $s3: 0x0
reg $s4: 0x0
reg $s5: 0x0
reg $s6: 0x0
reg $s7: 0x0
reg $t8: 0x0
reg $t9: 0x0
reg $k0: 0x0
reg $k1: 0x0
reg $gp: 0x0
reg $sp: 0xffffffff
reg $fp: 0x0
reg $ra: 0x0
```

Memory print:

```
Executing memory stage ...
writing to memory ...
contents of memory:
address 0x0: 0x0
address 0x1: 0x0
address 0x3: 0x0
address 0x2: 0x0
```

By the end of simulation, the total clock cycles elapsed are printed as shown below

```
Total clock cycles elapsed = 15
Program is Terminating... Bye Bye
```