UNIVERSITY OF THE PHILIPPINES
DILIMAN

CS 21

MACHINE PROBLEM

MIPS in MIPS for Dummies

Authors
ESPARTERO, Joshua Allyn Marck

Supervisor
BALINGIT, Ivan Carlo

June 20, 2021

Table of Contents

1. Introduction

1.1 Objective

This project aims to create a MIPS assembly program that will simulate a MIPS assembly environment.

1.2 Specifications

The project specifications state that we are to create a MIPS simulator implemented in MIPS assembly code.

Numeric limits and defaults

1. **Accessible memory addresses:** 0x00000 to 0xfffff (inclusive); only 1 MiB of main memory for the simulated program
2. **Data segment starting address**: 0x10000
3. **Stack pointer starting value**: 0x20000
4. **Number of instructions for input MIPS programs**: 2 to 100 (inclusive).

The project specifies fourteen (14) basic instructions to be implemented:

1. add
2. sub
3. and
4. or
5. addiu
6. slt
7. beq
8. bne
9. lw
10. sw
11. j

12. jal
13. jr
14. syscall

1.3 General Overview of Code Structure

This is the general flow of the MIPS code that was written by the author.

1. Allocate heap memory for the following:
    - Registers array
    - Memory of the simulation
2. Accept the instructions encoded in decimal
3. Read and Execute the instructions
    - While loop keep on converting instructions until exit is called
    - Bitmasking to grab parts of the instruction
    - Check funct to know which kind of instruction (R, I, J, or Jal)
    - Futher check which kind of R/I/J-type instruction
    - Handle each instruction accordingly
    - Pointer arithmetic to move through the instructions.

2 Helper Tools

2.1 .Eqvs

We first enumerate all the .eqvs used in the program. These are used to make the program more readable and helps the author make the code more intuitive.

First we have .eqvs used for register renaming:

1. $t9 - **N**
2. $t8 - **counter**
3. $t7 - **PC**
4. $t6 - **memory**
5. $t5 - **offset**
6. $t4 - **registers**
7. $t2 - **TEMP**
8. $t1 - **startmem**
9. $t0 - **instructions**
10. $s0 - **opcode**
11. $s1 - **rs**
12. $s2 - **rt**
13. $s3 - **rd**
14. $s4 - **imm**
15. $s5 - **funct**
16. $s6 - **beqRt**
17. $s7 - **printX**

Then we have .eqvs used for constants, these are constants used in bitmasking to grab certain parts of our instruction.

1. 0xFC000000 - **opcodeMask**
2. 0x30E00000 - **rsMask**

3. 0x001F0000 - **rtMask**
4. 0x0000F800 - **rdMask**
5. 0x3F - **functMask**
6. 0x3FFFFFF - **addressMask**
7. 0xFFFF - **immMask**

2.2 Macros

These macros are created to easily call out blocks of code that are used multiple times in the code or to make some parts of the main code more intuitive and readable.

1. do_syscall(%n) - used for a convenient way of calling syscalls.

```
.macro do_syscall(%n)
  li  $v0, %n
  syscall
.end_macro
```

2. allocateMemory(%n,%x) - allocates **%x** heap memory to the register **%n**.

```
.macro allocateMemory(%n, %x)
  li  $a0, %x
  do_syscall(9)
  move  %n, $v0
.end_macro
```

3. printInt(%d) - prints out the integer %d.

```
.macro printInt(%d)
  move  $a0, %d
  do_syscall(1)
.end_macro
```

4. printChar(%c) - prints out the character %c.

```
.macro printChar(%c)
  move  $a0, %c
  do_syscall(11)
.end_macro
```

---

3 System Design

3.1 PC Counter

$PC\ counter$ is implemented in a way where $PC \in [0, 1, 2, 3, ...]$ where the $PC = i$ means that we're currently processing the i-th instruction located in our simulator's .text segment.

This lets us get a way with multiple lines of code of converting Jump instructions to the actual addresses. We could just set PC equal to the address encoded in the instruction.

However, because of this we need to multiply the current PC counter by 4 in the *setNext* portion of our code to access the location of the next instruction since it is word-aligned.

Here's a snippet of the *setNext* segment where this is done:

```
setNext:
  ...
  sll offset, PC, 2       # PC = PC*4 (SINCE WORD ALIGNED INSTRUCTIONS ARRAY)
  add instructions, instructions, offset  # *INSTRUCTIONS = INSTRUCTIONS[PC]
  ...
```

3.2 Memory Allocation

1. Simulation Memory

   We allocate **0x100000 bytes** worth of heap memory to our simulation as needed. The memory variable then becomes a pointer containing the address of the first element of our simulation memory (0x000000). This is done with the help of the $allocateMemory$ macro discussed in the Helper Tools portion of this documentation. This can be found in the *start* segment of our code.

2. Registers Array

   We allocate 132 (4*32) bytes of worth of heap memory to our registers array to account for the 32 registers that we need for our simulation. The registers variable then becomes a pointer containing the address of our simulation's Zero Register (Register[0]). This is done with the help of the $allocateMemory$ macro discussed in the Helper Tools portion of this documentation. This can be found in the *start* segment of our code.

A snippet of the start segment of our code containing both the lines that allocate heap memory to our simulation memory and registers array.

```
start:  #START OF SIMULATION
  .....
  allocateMemory(registers, 132)
  allocateMemory(memory, 0x100000)  # set starting point of memorry
  .....
```

3.3 Accepting Input

First we accept the first line of input which is meant to be the number of instructions that the input wants us to simulate. This is stored in N. This is done by using syscall code 5. This is done in the start of our code at the start segment. A snippet is show below.

```
start:  #START OF SIMULATION
  do_syscall(5)       #ACCEPT INTEGER
  move  N, $v0
  ...
```

Then we accept the instruction **N** times and store it on the first part of our simulation's text segment starting from the simulation address 0x00000000 onwards. We first set the instruction variable to be a pointer pointing to the starting address of our memory simulation. Then we slowly populate our text segment with the input instructions one by one in a loop that will end when the counter is greater than or equal to N. After we reset the instructions pointer to the start of the instructions array.

Below is a snippet of the code and it contains additional explanations per line.

```
move  instructions, memory    # initialize instructions array pointer

inputRead:
  do_syscall(5)         # ACCEPT INTEGER
  move  $a0, $v0        # store the integer into $a0
  sw  $a0, (instructions)   # store integer to instructions[i] where i = [0,1,2,3...]

  addiu   instructions, instructions, 4 # move instructions pointer to next address
  addiu   counter, counter, 1   # add one to our counter variable
  bge   counter, N, inputRead   # if counter != N, continue accepting
  move  instructions, startmem    # reset instructions pointer to the instructions[0]
  li  counter, 0      # reset counter variable
```

3.4 Reading Opcode

After we have finished reading the inputs, we now proceed into reading the instructions one by one and figuring out what kind of instruction it is. We do this by loading the instruction that is handled by our PC counter into $a0. We then perform bit-masking that will let us grab the opcode (instruction[31:26]). Here we use the mask, $0xFC000000$ which is conveniently named opcodeMask in our .eqvs portion. We perform an andi operation with the instruction and store the result in our opcode. Note that we did not perform SRL anymore since the author thought that we can still figure out what kind of instruction it is even if the bits are just hanging at the 31st to 26th bits of our instruction.

We then check which type of instruction it is, R-Type/I-Type/J/Jal/syscall. We do this by comparing the opcode/instruction to certain constants.

1. syscall - we can check if it is a syscall if the instruction itself is just 0xC or (12). We then branch to where syscalls will be handled in our code.
2. R-Type - we can check if it is an R-Type if the opcode is equal to 0. We then branch to where R-Types will be handled in our code.

3. J (jump) - we can check if it is a J instruction if the opcode is equal to $0x8000000$ or $'b000010$ is the content of the funct field of the instruction . We then branch to where J instruction will be handled in our code.
4. Jal - we can check if it is a Jal instruction if the opcode is equal to 0. We then branch to where Jal instructions will be handled in our code.
5. I-Type - if none of the other conditions for the other cases are met, we consider this a J-Type instruction. We then jump to where I-Type instructions are handled in our code.

These are located in the *convert*: segment of our code.

Here is a snippet of the lines of code for Reading the Opcode.

```
convert:
  #INSTRUCTION LOADING
  lw  $a0, (instructions)   # store the i-th instruction in $a0
  andi  opcode, $a0, opcodeMask   # do bitmasking to retrieve the opcode

  #TYPE CHECKING        # we check what kind of instruction it is
  beq   $a0, 0xC, sys     # if $a0 = 12, then go to syscall
  beq   opcode, 0, rtype    # if opcode = 0, go to R TYPE HANDLING
  beq   opcode, 0x8000000, jtype  # if opcode = 0x8000000, go to J PROCEDURE
  beq   opcode, 0xC000000, j1   # if opcode = 0xC000000, go to JAL PROCEDURE
  j   itype       # else, it is a j-type instruction
```

3.5 Syscall Handling

After we encounter an instruction "$0xC$" we proceed to handling syscalls in our code. We will encounter 3 kinds of syscalls that are specified in our machine problem specifications.

1. Print Integer (syscall code 1)
2. Exit (syscall code 10)
3. Print Character (syscall code 11)

We first check the contents of our simulation register $v0 by loading registers[2] of our registers array. We then branch to the portion where we handle each kind of syscall accordingly.

```
sys:  #SYSCALL PROCEDURE
  lw  TEMP, 8(registers)    # load registers[2] ($v0)
  beq   TEMP, 1, s1     # if $v0 = 1: go to PRINT INTEGER (s1)
  beq   TEMP, 10, s2      # if $v0 = 10: go to EXIT(0) (s2)
  beq   TEMP, 11, s3      # if $v0 = 11: go to PRINT CHARACTER (s3)
  j   while       # else go to while (next instruction)
```

1. **Print Integer** - this will be executed by loading the contents of our simulation register $a0 which is done by loading registers[4] we print this out with the help of our $printInt$ macro. We then proceed to the end part of our code to load a new instruction.

    ```
    s1: #PRINT INTEGER PROCEDURE
      lw  printX, 16(registers)   # load registers[4] ($a0)
      printInt(printX)      # call macro printInt
      j   while       # go to while (next instruction)
    ```

2. **Exit** - this will be executed by just performing a real syscall code 10 in our code. This is done with the help of our $do\_syscall$ macro.

    ```
    s2: #EXIT(0) PROCEDURE
      do_syscall(10)        # call syscall 10 using do_syscall macro
    ```

3. **Print Character** - this will be executed by loading the contents of our simulation register $a0 which is done by loading registers[4] we print this out with the help of our $printChar$ macro. We then proceed to the end part of our code to load a new instruction.

    ```
    s3: #PRINT CHARACTER PROCDEDURE
      lw  printX, 16(registers)   # load registers[4] ($a0)
      printChar(printX)     # call macro printChar
      j while       # go to while (next instruction)
    ```

3.6 R-Type Instruction Handling

We now discuss on how do we handle R-Type instructions in our code.

Before we proceed into handling specific R-Types we must first find out what do our instruction's specific fields contain. We do this by performing bit-masking for every field and storing the results to different variables conveniently named to be descriptive enough of what it is containing.

We use the following masks for the different fields:

1. rs field - 0x03E00000 (.eqv rsMask)
2. rt field - 0x001F0000 (.eqv rtMask)
3. rd field - 0x0000F800 (.eqv rdMask)
4. funct field - 0x3F (.eqv functMask)

Here is a snippet of the code that does this:

```
rtype:  # R TYPE HANDLING

  # MASKING AND PREPROCESSING
  andi  rs, $a0, rsMask     # grab rs using rsMask
  andi  rt, $a0, rtMask     # grab rt using rtMask
  andi  rd, $a0, rdMask     # grab rd using rdMask
  andi  funct, $a0, functMask   # grab funct using functMask
```

We then perform bit-shifting to remove the trailing 0s that we dont need, while doing this we look ahead to what we will do next: We need to multiply the **rs**, **rt**, and **rd** fields by 4 since we need to access them in word-aligned array.

Here is the snippet of the code on how this was implemented:

```
# BIT-SHIFTING WITH *4 LOOK-AHEAD FOR ACCESSING REGISTERS (WORD-ALIGNED)
  srl   rs, rs, 19    # 21-2 move 21 to right then move 2 to the left (times 4)
  srl   rt, rt, 14    # 16-2 move 16 to right then move 2 to the left (times 4)
  srl   rd, rd, 9     # 11-2 move 11 to right then move 2 to the left (times 4)
```

Then we set the rs, rt, and rd fields as pointers to their corresponding registers[rs/rt/rd] by adding the values we achieved from the bit-shifting with times 4 look ahead portion to the pointer of our registers array.

Here is the snippet of the code on how this was implemented:

```
# POINTER ARITHMETIC TO SET RS, RT, AND RD AS POINTERS TO THEIR CORRESPONDING REGI
STERS
  add   rs, rs, registers
  add   rt, rt, registers
  add   rd, rd, registers
```

We then deference RS and RT while RD stays as a pointer since we need the actual values of the corresponding registers of our RS and RT fields while RD will be where we will store the result of the R-Type instruction.

Here is a snippet of the block of code on how this was implemented:

```
# DEREFERENCE RS AND RT
  lw  rs, (rs)
  lw  rt, (rt)
  # NOTE: RD STAYS AS A POINTER
```

Then we go to checking what specific R-Type instruction we are dealing with by looking at the funct field of the instruction:

Note: We also need to take into account if we're trying to edit the $0 (so it should proceed to the next instruction if ever it does).

Here is a snippet of the block of code on how this was implemented and some explanation on each case:

```
# CHECK R-TYPE INSTRUCTION TYPE
  beq   funct, 0x08, j2     # IF FUNCT = b'001000: go to JR PROCEDURE
  beq   rd, 0, while        # IF TRYING TO WRITE TO $0 REGISTER, SKIP
  beq   funct, 0x20, r1     # IF FUNCT = b'100000: go to ADD PROCEDURE
  beq   funct, 0x22, r2     # IF FUNCT = b'100010: go to SUB PROCEDURE
  beq   funct, 0x24, r3     # IF FUNCT = b'100100: go to AND PROCEDURE
  beq   funct, 0x25, r4     # IF FUNCT = b'100101: go to OR PROCEDURE
  beq   funct, 0x2A, r5     # IF FUNCT = b'101010: go to SLT PROCEDURE
  j   while       # go to while ( if none: next instruction )
```

We then proceed into handling the specific R-Type Instructions:

1. **ADD** - we just store the sum of actual values of register[rs] and register[rt] in our TEMP variable. Then we store in into registers[rd].

   ```
   r1:   # ADD PROCEDURE
     add   TEMP, rs, rt      # ADD RS AND RT
     sw  TEMP, (rd)          # STORE SUM INTO ADDRESS OF RD REGISTER
     j   while               # go to while (next instruction)
   ```

2. **SUB** - we just store the difference of actual values of register[rs] and register[rt] in our TEMP variable. Then we store in into registers[rd].

   ```
   r2:   # SUB PROCUDURE
     sub   TEMP, rs, rt      # SUBTRACT RT from RS
     sw  TEMP, (rd)      # STORE DIFFERENCE INTO ADDRESS OF RD REGISTER
     j   while       # go to while
   ```

3. **AND** - we just store the result of actual values of register[rs] & register[rt] in our TEMP variable. Then we store in into registers[rd].

   ```
   r3:   # AND PROCEDURE
     and   TEMP, rs, rt      # PERFORM BIT-WISE AND ON RS AND RT
     sw  TEMP, (rd)      # STORE OUTPUT INTO ADDRESS OF RD REGISTER
     j   while       # go to while
   ```

4. **OR** - we just store the result of actual values of register[rs] | register[rt] in our TEMP variable. Then we store in into registers[rd].

   ```
   r4:   # OR PROCEDURE
     or  TEMP, rs, rt      # PERFORM BIT-WISE OR ON RS AND RT
     sw  TEMP, (rd)      # STORE OUTPUT INTO ADDRESS OF RD REGISTER
     j   while       # go to while
   ```

5. **SLT** - we just perform an actual slt instruction with the values of the registers of rs and rt and store it into registers[rd].

   ```
   r5:   # SLT PROCEDURE
     slt TEMP, rs, rt      # IF RS < RT ? 1 : 0
   ```

```
  sw  TEMP, (rd)      # STORE OUTPUT INTO ADDRESS OF RD REGISTER
  j   while       # go to while
```

6. JR - we just set our PC to the content of registers[rs]. However, note that because of our implementation of the PC counter we need to divide the address by 4. Thus, we shift it twice to the right.

```
j2:   # JR PROCEDURE
  srl TEMP, rs, 2
  move  PC, TEMP      # SET PC TO $rs
  j   setNext         # read next instruction in setNext
```

After each procedure, we just jump to either the while portion of our code if we need to do PC + 1 to load the next instruction or jump to the setNext portion of our code if we just need to directly load the next instruction (i.e. if we performed JR).

3.7 Handling I-Type Instructions

We now discuss on how do we handle J-Type instructions in our code.

Before we proceed into handling specific J-Types we must first find out what do our instruction's specific fields contain. We do this by performing bit-masking for every field and storing the results to different variables conveniently named to be descriptive enough of what it is containing.

We use the following masks for the different fields:

1. rs field - 0x03E00000 (.eqv rsMask)
2. rt field - 0x001F0000 (.eqv rtMask)
3. imm field - 0xFFFF (.eqv rdMask)

Here is the snipper of the part of our code that does this:

```
itype:  # I TYPE HANDLING
```

```
  # MASKING AND PREPROCESSING
  andi  rs, $a0, rsMask     # GRAB RS USING rsMask
  andi  rt, $a0, rtMask     # GRAB RT USING rtMask
  andi  imm, $a0, immMask   # GRAB IMM USING immMask
```

We sign-extend our imm. We check if its 16th bit is 1. We do this by checking the imm is greater the 0x8000. If so then we set the upper 16 bits to be all 1's. Otherwise, leave them as is (all 0's). We perform this logic in this part of our code:

```
# CHECKING FOR NEGATIVE IMMs
  bge   imm, 0x00008000, immNeg   # IF LEADING BIT OF IMM = 'b1
  j   iB        # ELSE PROCEED

immNeg: # SIGNEXTENDING NEGATIVE IMMs
  ori   imm, imm, 0xFFFF0000    # PERFORMING BITWISE OR TO EXTEND NEGATIVE IMM
  j   iB        # PROCEED

iB:
  ...
```

We then perform bit-shifting to remove the trailing 0s that we dont need, while doing this we look ahead to what we will do next: We need to multiply the **rs** and **rt** fields by 4 since we need to access them in word-aligned array.

Here is the snippet of the part of the code that does this:

```
iB:
  # BIT-SHIFTING WITH *4 LOOK-AHEAD FOR ACCESSING REGISTERS (WORD-ALIGNED)
  srl   rs, rs, 19  # 21-2 move 21 to right then move 2 to the left (times 4)
  srl   rt, rt, 14  # 16-2 move 16 to right then move 2 to the left (times 4)
  ...
```

We then dereference RS while RT stays as a pointer as we need the value in registers[rs] while RT

will be where we will store the result of the I-Type instruction. However, for the BEQ instruction we make a special case since we actually need to compare both RS and RT. So we store the registers[rt] in a variable named **beqRt**.

Here is the snippet of the part of the code that does this:

```
# DERERENCING POINTERS TO ACCESS VALUES
  lw  rs, (rs)      # SETS RS TO BE THE DEREFERENCED RS
  lw  beqRt, (rt)   # SETS BEQRT TO BE THE DEREFERENCED RT (TO BE USED IN BEQ)
                      # NOTE: BEQRT != RT; RT IS STILL NOT DEREFERENCED
```

Then we go to checking what specific I-Type instruction we are dealing with by looking at the opcode field of the instruction:

> Note: We also need to take into account if we're trying to edit the $0 (so it should proceed to the next instruction if ever it does).
>
> Here is a snippet of the block of code on how this was implemented and some explanation on each case:

```
# I-TYPE INSTRUCTION TYPE CHECKING
  beq   rt, 0, while      # IF TRYING TO EDIT $0 THEN PASS
  beq   opcode, 0x24000000, i1    # IF OPCODE = 0x24000000: PROCEED TO ADDIU PROCE
DURE
  beq   opcode, 0x10000000, i2    # IF OPCODE = 0x10000000: PROCEED TO BEQ PROCEDU
RE
  beq   opcode, 0x8c000000, i3    # IF OPCODE = 0x8c000000: PROCEED TO LW PROCEDUR
E
  beq   opcode, 0xac000000, i4    # IF OPCODE = 0x8c000000: PROCEED TO SW PROCEDUR
E
  beq opcode, 0x14000000, i5    # IF OPCODE   : PROCEED TO BNE PROCEDURE
  j   while       # else go to while (invalid I-Type) Next instruction
```

We then proceed into handling the specific I-Type Instructions:

1. **ADDIU** - we just perform a normal addu operation with the contents of registers[rs] and the sign extended imm of our instruction.

   ```
   i1:   # ADDIU PROCEDURE
     addu  TEMP, rs, imm     # ADD RS AND SIGNEDIMM
     sw  TEMP, 0(rt)     # STORE SUM INTO REGISTERS[RT]
     j   while       # go to while
   ```

2. **BEQ** - we check if registers[rs] ≠ registers[rd], then we go to the next instruction (no branching will be done). Else, we set the PC counter to the sign-extended imm of our instruction. However, it should be PC+1 (+4) + the

imm. So it will be handled in the latter part of the code (in the while segment).

```
i2:   # BEQ PROCEDURE
  bne   rs, beqRt, while    # IF RS != BEQRT, NO BRANCHING WILL BE DONE
  add   PC, PC, imm     # ELSE: ADD IMM TO PC (+1 will handled in while)
  j   while       # GO TO WHILE
```

3. **LW** - we add the offset to the contents of our register[rs]. Then we set the result relative to the start of our simulation memory. Then we load that part of the memory to our TEMP variable then we store it to registers[rt].

```
i3:   # LW PROCEDURE
  add   rs, rs, imm     # ADDING OFFSET TO THE ADDRESS IN RS
  add   rs, rs, memory      # SETTING RELATIVE OFFSET
  lw  TEMP, (rs)      # LOADING THE ADDRESS WE'VE CALCULATED
  sw  TEMP, (rt)      # STORING DATA INTO REGISTERS[RT]
  j while       # go to while
```

4. **SW** - we add the offset to the contents of our register[rs]. Then we set the result relative to the start of our simulation memory. Then we load the value in registers[rt] and then we store it to the resulting address of the relative offset we obtained earlier.

```
i4:   # SW PROCEDURE
  add   rs, rs, imm     # ADDING OFFSET TO THE ADDRESS IN RS
  add   rs, rs, memory      # SETTING RELATIVE OFFSET
  lw  TEMP, (rt)      # LOADING DATA FROM REGISTER[RT]
  sw  TEMP, (rs)      # STORING DATA IN TO THE ADDRESS WE'VE CALCULATED
  j   while       # go to while
```

5. **BNE** - we check if registers[rs] = registers[rd], then we go to the next instruction (no branching will be done). Else, we set the PC counter to the sign-extended imm of our instruction. However, it should be PC+1 (+4) + the imm. So it will be handled in the latter part of the code (in the while segment).

```
i5:   # BNE PROCEDURE
  beq   rs, beqRt, while    # IF RS != BEQRT, NO BRANCHING WILL BE DONE
  add   PC, PC, imm     # ELSE: ADD IMM TO PC (+1 will handled in while)
  j   while       # GO TO WHILE
```

3.8 Handling Jump and Jal Instructions

Earlier, when we checked for the opcodes we had special branches for Jal and Jump. In this section we discuss how do we execute those instructions in our simulator.

1. **Jump** - we just set the PC counter to the offset address that the instruction contains (which we grab by performing bitwise and with addressMask - 0x3FFFFFF). We handle the word-aligned instructions problem in the setNext portion of our code where we shift the PC variable twice to the left.

   ```
   jtype:  # J PROCEDURE
       andi  PC, $a0, addressMask  # GRAB THE ADDRESS FROM THE INSTRUCTION
                                   # USING addressMask
       j   setNext       # handle the bit shifting in setNext
   ```

2. **Jal** - We calculate (PC+1) to signify the address offset of our next instruction. Then we store that to registers[31] ($ra). We then set the PC counter to the offset address that the instruction contains (which we grab by performing bitwise and with addressMask - 0x3FFFFFF). We handle the word-aligned instructions problem in the setNext portion of our code where we shift the PC variable twice to the left.

   ```
   j1:   # JAL PROCEDURE
     addi  TEMP, PC, 1     # CALCULATE (PC+1)4
     sll TEMP, TEMP, 2     #
     sw  TEMP, 124(registers)    # THEN STORE AT REGISTERS[31] ($ra)
     andi  PC, $a0, addressMask    # GRAB ADDRESS OF FUNCTION USING addressMarsk
     j   setNext       # handle bit shifting in setNext
   ```

3.9 Reading the Next Instruction

At the latter part of our code, we have the while segment that handles the PC + 1 to access the next instruction. This is coupled with the setNext segment that converts the PC counter to the word-aligned offset that we need for us to access the next instruction. Note that not all instructions need the PC + 1. Jump Type instructions directly set the PC to the address of the next instruction so they directly go to the setNext segment and skip the while segment:

Here is the snippet of code that handles the logic needed for this:

```
while:
  addiu   PC, PC, 1     # PC = PC + 1

setNext:
  move  instructions, startmem      # *INSTRUCTIONS = INSTRUCTIONS[0]
  sll offset, PC, 2       # PC = PC*4 (SINCE WORD ALIGNED INSTRUCTIONS ARRAY)
  add instructions, instructions, offset  # *INSTRUCTIONS = INSTRUCTIONS[PC]
  j convert       # WHILE PC in TEXT SEGMENT KEEP ON CONVERTING
```

This marks the end of this documentation. Thank you for reading my documentation.