**CSCE 5610 – Computer System Architecture**

Assignment 1

Uday Bhaskar Valapadasu

11696364

**1Ans:**

Given,

* Instruction Length – 14 bits
* 64-bit general purpose registers i.e. each register is of size 64- bits.
* Address Field – 6 bits i.e. ( one address field can get 26 combinations.)

Instructions which can be encoded within the 14-bit instruction length: **214 =** **16,384** possible outcomes.

**1.a) Check if the below instructions are possible to encode:**

3 two-address instructions

63 one-address instructions

45 zero-address instructions

**Calculating the possible outcomes the different instructions:**  
3 two-address instructions = 3 x 26 x 26 = 12,288

63 one-address instructions = 63 x 26 = 4,032  
45 zero-address instructions = 45 x 20 = 45  
Total = 12,288 + 4,032+45 = 16,365

Therefore, we can observe the 16,365 < 16,384.

Yes, it is possible to encode all the instructions given.

**1.b)** **Check if the below instructions are possible to encode:**

3 two-address instructions  
65 one-address instructions  
35 zero-address instructions

**Calculating the possible outcomes for the different instructions:**

3 two-address instructions = 3 x 26 x 26 = 12,288

65 one-address instructions = 65 x 26 = 4,160

35 zero-address instructions = 35 x 20 = 35

Total = 12,288 + 4,160 + 35 = 16,483

Therefore, we can observe that 16,483 > 16,384

No, it is not possible to encode all the instructions given.

**1.c) Check if the below instructions are possible to encode:**

3 two-address instructions  
24 zero-address instructions

**Calculating the outcomes for the existing instructions**:

3 two-address instructions = 3 x 26 x 26 = 12,288

24 zero-address instructions = 24 x 20 = 24

Total for existing instructions = 12,288 + 24 = 12,312

**Calculating remaining outcomes for one-address instructions**:

Remaining outcomes=16,384 − 12,312 = 4,072

**Maximum number of one-address instructions**:

Each one-address instruction can encode one address (6 bits):

Maximum one-address instructions=4,072 / 26 = 4,072 / 64 = 63.25

Since we can't have a fraction of an instruction.

Thus, the maximum number of one-address instructions that can be encoded is **63**.

**2Ans: C code snippet to MIPS assembly code:**

**2.1)** Given,$t0 = a, $t1 = b, $s0 = base address of array c

# a = a - b

sub $t0, $t0, $t1 # Compute a - b and store the result back in $t0

# Set c[0] to b + 1

addi $t2, $t1, 1 # Increment b by 1 and store the result in $t2

sw $t2, 0($s0) # Store the value in $t2 at the memory location for c[0]

# Compute c[a + 2] as c[0] - c[b + 1]

lw $t3, 0($s0) # Load the value of c[0] into $t3

addi $t4, $t1, 1 # Calculate b + 1 and store it in $t4

sll $t4, $t4, 2 # Multiply the index (b + 1) by 4 (size of int) for byte offset

add $t4, $t4, $s0 # Adding the offset with base address.

lw $t5, 0($t4) # Load the value of c[b + 1] into $t5

addi $t6, $t0, 2 # Calculate a + 2 and store it in $t6

sll $t6, $t6, 2 # Multiply the index (a + 2) by 4 (size of int) for byte offset

add $t6, $t6, $s0 # Adding the offset with base address

sub $t7, $t3, $t5 # Subtract c[b + 1] (in $t5) from c[0] (in $t3) and store in $t7

sw $t7, 0($t6) # Store the result from $t7 at the memory location for c[a + 2]

**2.2)** Given,$t0 = a, $t1 = b, $t2 = c, $s0 = base address of array d

**2.2.a)**

# Compare b and c

ble $t1, $t2, if\_part # If b <= c, branch to if\_part

# Else part (b > c)

sll $t3, $t1, 1 # $t3 = b \* 2 (multiply b by 2 using left shift)

add $t3, $t0, $t3 # $t3 = a + (b \* 2)

sll $t4, $t1, 2 # $t4 = b \* 4 (multiply b by 4 for array index)

add $t4, $s0, $t4 # $t4 = &d[b] (calculate address of d[b])

sw $t3, 0($t4) # d[b] = a + b \* 2 (store result in d[b])

j end\_if # Jump to end of if statement

# If part (b <= c)

if\_part:

srl $t3, $t0, 2 # $t3 = a / 4 (divide a by 4 using right shift)

sll $t4, $t1, 2 # $t4 = b \* 4 (multiply b by 4 for array index)

add $t4, $s0, $t4 # $t4 = &d[b] (calculate address of d[b])

sw $t3, 0($t4) # d[b] = a / 4 (store result in d[b])

end\_if:

# End of if-else statement

**2.2.b)**

# Store result in d[i]

sw $t6, 0($t3) # Write result to d[i]

# Increment i

addi $t2, $t2, 1 # Increase i by 1

# Jump back to start of loop

j loop\_start

loop\_end:

# End of for loop

# Initialize i to 0

li $t2, 0 # Set i = 0

# Load d[b] once before the loop

sll $t3, $t1, 2 # Compute b \* 4 (index for d[b])

add $t3, $s0, $t3 # Get address of d[b]

lw $t4, 0($t3) # Load d[b] into $t4

loop\_start:

# Check loop condition (i < a)

bge $t2, $t0, loop\_end # If i >= a, exit the loop

# Calculate address of d[i]

sll $t3, $t2, 2 # Compute i \* 4 (index for d[i])

add $t3, $s0, $t3 # Get address of d[i]

# Calculate 2 \* i

sll $t5, $t2, 1 # Compute 2 \* i

# Compute d[b] - 2 \* i

sub $t6, $t4, $t5 # Calculate d[b] - 2 \* i

**2.2.c)**

# Initialize i to 0

li $t1, 0 # $t1 = i = 0

loop\_start:

# Calculate address of d[i]

sll $t2, $t1, 2 # $t2 = i \* 4 (multiply i by 4 for array index)

add $t2, $s0, $t2 # $t2 = &d[i] (address of d[i])

# Load d[i] and check if it's > 0

lw $t3, 0($t2) # $t3 = d[i]

blez $t3, loop\_end # If d[i] <= 0, exit loop

# Calculate d[a] + 4 \* i

sll $t4, $t1, 2 # $t4 = 4 \* i (multiply i by 4 using left shift)

sll $t5, $t0, 2 # $t5 = a \* 4 (multiply a by 4 for array index)

add $t5, $s0, $t5 # $t5 = &d[a] (address of d[a])

lw $t5, 0($t5) # $t5 = d[a]

add $t4, $t5, $t4 # $t4 = d[a] + 4 \* i

# Store result back in d[i]

sw $t4, 0($t2) # d[i] = d[a] + 4 \* i

# Increment i

addi $t1, $t1, 1 # i++

# Jump back to start of loop

j loop\_start

loop\_end:

# End of while loop

**2.2.c)**

**2.3) Fibonacci function -** We define n in the .data section and initialize it to 3.

fib:

# sstack frame setup

addi $sp, $sp, -12 # Allocate space on the stack

sw $ra, 8($sp) # Save return address

sw $s0, 4($sp) # Save $s0 for storing n

sw $s1, 0($sp) # Save $s1 for calculations

move $s0, $a0 # Moving n to $s0

# if (n <= 1) return n

ble $s0, 1, fib\_base\_case

# This steps involves recursion

addi $a0, $s0, -1 # Calculate n-1

jal fib # Recursive call fib(n-1)

move $s1, $v0 # Store fib(n-1) result in $s1

addi $a0, $s0, -2 # Calculate n-2

jal fib # Recursive call fib(n-2)

add $v0, $s1, $v0 # $v0 = fib(n-1) + fib(n-2)

j fib\_return

fib\_base\_case:

move $v0, $s0 # Return n for base case

fib\_return:

# Restore stack frame

lw $s1, 0($sp) # $s1 restored

lw $s0, 4($sp) # $s0 restored

lw $ra, 8($sp) # return address restored

addi $sp, $sp, 12 # Free up stack space

jr $ra # caller returned

main:

# Set up stack frame

addi $sp, $sp, -4 # Allocate space on the stack

sw $ra, 0($sp) # Save return address

lw $a0, n # Load n into $a0

jal fib # Call fib(n)

# The result is now in $v0, but not used here

# Restore stack frame

lw $ra, 0($sp) # Restore return address

addi $sp, $sp, 4 # Free up stack space

li $v0, 10 # Prepare for exit system call

syscall # Exit program

**3Ans**

**Step 1:** Convert each and every hexadecimal instruction to binary.

|  |  |
| --- | --- |
| 0X20080000 | 0010 0000 0000 1000 0000 0000 0000 0000 |
| 0X11090006 | 0001 0001 0000 1001 0000 0000 0000 0110 |
| 0X00085042 | 0000 0000 0000 1000 0101 0000 0100 0010 |
| 0X00085880 | 0000 0000 0000 1000 0101 1000 1000 0000 |
| 0X01705820 | 0000 0001 0111 0000 0101 1000 0010 0000 |
| 0XAD6A0000 | 1010 1101 0110 1010 0000 0000 0000 0000 |
| 0X21080001 | 0010 0001 0000 1000 0000 0000 0000 0001 |
| 0X08001001 | 0000 1000 0000 0000 0001 0000 0000 0001 |

**Step 2:** Decode each instruction using the MIPS instruction format:

**Register Table:**

|  |  |  |
| --- | --- | --- |
| **Register Name** | **Register Number** | **Binary Value** |
| $zero | 0 | 00000 |
| $t0 | 8 | 01000 |
| $t1 | 9 | 01001 |
| $t2 | 10 | 01010 |
| $t3 | 11 | 01011 |
| $s0 | 16 | 10000 |
| $t2 | 10 | 01010 |

**Opcode and Function Code Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Format** | **Opcode (6 bits)** | **Function (6 bits)** |
| addi | I-type | 001000 (8) | N/A |
| beq | I-type | 000100 (4) | N/A |
| srl | R-type | 000000 (0) | 000010 (2) |
| sll | R-type | 000000 (0) | 000000 (0) |
| add | R-type | 000000 (0) | 100000 (32) |
| sw | I-type | 101011 (43) | N/A |
| j | J-type | 000010 (2) | N/A |

**Decoding Table (Separating Instruction Based on Its Type Format):**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Hex Instruction** | **Type** | **Opcode (6 bits)** | **rs (5 bits)** | **rt (5 bits)** | **rd (5 bits)** | **shamt (5 bits)** | **funct (6 bits)** | **Immediate (16 bits) / Target (26 bits)** |
| 0x20080000 | I | 001000 | 00000 | 01000 | N/A | N/A | N/A | 0000 0000 0000 0000 |
| 0x11090006 | I | 000100 | 01000 | 01001 | N/A | N/A | N/A | 0000 0000 0000 0110 |
| 0x00085042 | R | 000000 | 00000 | 01000 | 01001 | 00001 | 100010 | N/A |
| 0x00085880 | R | 000000 | 00000 | 01000 | 01011 | 00010 | 000000 | N/A |
| 0x01705820 | R | 000000 | 01011 | 10000 | 01000 | 00000 | 100000 | N/A |
| 0xAD6A0000 | I | 101011 | 01011 | 01010 | N/A | N/A | N/A | 0000 0000 0000 0000 |
| 0x21080001 | I | 001000 | 01000 | 01000 | N/A | N/A | N/A | 0000 0000 0000 0001 |
| 0x08001001 | J | 000010 | N/A | N/A | N/A | N/A | N/A | 0000 0000 0000 1000 0000 0000 0001 |

**Decoding Table(In Decimal Format):**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Hex Instruction** | **Type** | **Opcode** | **rs** | **rt** | **rd** | **shamt** | **funct** | **Immediate** | **Target** |
| 0x20080000 | I-type | 8 | 0 | 8 | - | - | - | 0 | - |
| 0x11090006 | I-type | 4 | 8 | 9 | - | - | - | 6 | - |
| 0x00085042 | R-type | 0 | 0 | 8 | 10 | 1 | 2 | - | - |
| 0x00085880 | R-type | 0 | 0 | 8 | 11 | 2 | 0 | - | - |
| 0x01705820 | R-type | 0 | 11 | 16 | 11 | 0 | 32 | - | - |
| 0xAD6A0000 | I-type | 43 | 11 | 10 | - | - | - | 0 | - |
| 0x21080001 | I-type | 8 | 8 | 8 | - | - | - | 1 | - |
| 0x08001001 | J-type | 2 | - | - | - | - | - | - | 4097 |

**MIPS instruction Table( Each Decoded Instruction in MIPS with its address ):**

|  |  |
| --- | --- |
| 0x00004000 | addi $t0, $zero, 0 |
| 0x00004004 | beq $t0, $t1, Exit |
| 0x00004008 | srl $t2, $t0, 1 |
| 0x00004012 | sll $t3, $t0, 2 |
| 0x00004016 | add $t3, $t3, $s0 |
| 0x00004020 | sw $t2, 0($t3) |
| 0x00004024 | addi $t0, $t0, 1 |
| 0x00004028 | j 0x001001 |

addi $t0, $zero, 0 # Initialize loop counter i = 0

loop:

beq $t0, $t1, end # If i == n, exit loop

srl $t2, $t0, 1 # $t2 = i / 2

sll $t3, $t0, 2 # $t3 = i \* 4 (byte offset)

add $t3, $t3, $s0 # $t3 = address of c[i]

sw $t2, 0($t3) # c[i] = i / 2

addi $t0, $t0, 1 # i++

j loop # Jump back to start of loop

end:

**Step 3: Assembly code:**

**Step 4:** This code implements a loop that iterates over the array c and sets each element to half of its index.

**Algorithms:**

* Set the loop counter i (stored in $t0) to 0.
* Compare i with n (the total number of elements). If they are equal, exit the loop.
* Compute i / 2 by performing a right shift (which is equivalent to dividing by 2).
* Determine the byte offset for c[i] by multiplying i by 4 (using a left shift).
* Find the address of c[i] by adding the offset to the base address of the array.
* Save the result of i / 2 into c[i].
* Increase i by 1.
* Loop back to the beginning.

C pseudocode, this would be equivalent to:

int i = 0;

while (i != n) {

c[i] = i / 2;

i++;

}