1. **MIPS Instructions**  

Consider the sequence of four instructions shown below.

loop:  
  lw $6, 0($5)  
  add $7, $7, $6  
  addi $5, $5, -4  
  bne $5, $8, loop

In the space provided below, show the binary representation of each of the four instructions in this sequence. Clearly mark the instruction fields and show their decimal equivalent.

- **lw $6, 0($5)**

  

- **add $7, $7, $6**

  

- **addi $5, $5, -4**

  

- **bne $5, $8, loop**

  

This exam is open book and open notes. You have 50 minutes. Credit for problems requiring calculation will be given only if you show your work.
2. MIPS Branch and Jump Instructions  

(a) The MIPS branch instructions are \texttt{beq} and \texttt{bne} can only change the current program counter by a limited amount. How many instructions can be reached using a branch instruction? What range of instructions can be reached relative to the program counter?

Since the offset is 15 bits, the beq and bne instructions can reach a total of $2^{16}$ instructions, including itself.

The range of instructions relative to the program counter goes from

\[(PC + 1 - 32,768) \text{ instructions backward}\]

to

\[(PC + 1 + 32,767) \text{ instructions forward}\]

(b) The MIPS jump (\texttt{j}) instruction is also limited in its ability to change the program counter. How many instructions can be reached using a jump instruction? What range of instructions can be reached relative to the program counter?

Since the offset is 26 bits, the beq and bne instructions can reach a total of $2^{26}$ instructions, including itself.

The range of instructions relative to the program counter is all instructions with address bits 31:28 matching \(PC[31:28]\).

(c) Given the limitations of the branch and jump instructions, we would like to create a pseudoinstruction that will allow a program to jump to any 32-bit address. Write a minimum-length sequence of MIPS instructions that implements this "jump anywhere" (ja) pseudoinstruction.

\begin{align*}
\text{Pseudoinstruction:} & \quad \text{Assembly Language:} \\
\texttt{ja address} & \quad \texttt{lui }$at, \text{ address-upper} \\
& \quad \texttt{ori }$at, $at, \text{ addr-lower} \\
& \quad \texttt{jr }$at
\end{align*}
3. Procedures and Functions 20 Points

The code fragment below shows a simple C function that is called by the main program. Note that this function is a leaf procedure i.e., it calls no other functions or procedures.

C Code

```c
int f(int a0, int a1) {
    return (a0 + a1);
}
```

Assembly Code

```assembly
f: add $v0, $a0, $a1
    jr $ra
```

```assembly
main:
    lui $s0, $s0, 1
    ori $s0, $s0, 30,464
    addi $s1, $s1, -12
    add $a0, $s0, $zero
    add $a1, $s1, $zero
    jal f
    add $s2, $v0, $zero
```

(a) Assume that variables s0, s1, and s2 and arguments a0 and a1 are stored in MIPS registers with the same name. In the space to the right of the C code above, fill in the MIPS assembly instructions that will implement the main function and the function f while storing arguments and changing the flow of control.

(b) Suppose that f is not a leaf procedure, but instead calls other leaf procedure g, which has three integer arguments. Briefly describe how your code would need to be changed. You do not need to write any code for this part – just describe what needs to be done, and list any values that would need to be stored on the stack.

Before f() calls g(), the argument registers $a0 and $a1 and the return address register $ra must be stored on the stack.

In general, the "saved registers" $s0-$s7 must also be stored across procedure calls if the calling procedure changes them (assume g doesn’t need to do this).

On return, values need to be restored and sp returned to its old value.
4. Carry Lookahead  

In a carry-lookahead adder, each bit-slice of the adder must use inputs \( a_i \) and \( b_i \) and \( c_i \) to produce generate function \( g_i \), propagate function \( p_i \), and sum output \( s_i \).

A proposal has been made to replace the standard definition of the propagate function:

\[
p_i = a_i + b_i
\]

With a version that uses the exclusive or function:

\[
p_i = a_i \oplus b_i
\]

The carry function is now implemented in the carry-lookahead unit as before:

\[
c_{i+1} = g_i + p_i \cdot c_i
\]

(a) Will the modified propagate function work properly in the calculation of carry signals by the carry lookahead unit? Are any additional modifications needed? Why or why not?

The only difference in \( p_i \) is that it will not be high when \( a_i \) and \( b_i \) are both high. However, in this case \( c_{i+1} \) will be 1 anyway because \( g_i \) will be 1, so the carry is not affected by the change. Yes, it will work.

(b) What advantage would there be in using the exclusive-or version of the propagate signal?

The advantage is that \( p_i \) can now also be used to help compute the sum of each bit (recall \( s_i = a_i \oplus b_i \oplus c_i \)). This can be a substantial savings for a 32-bit adder.

5. Verilog Coding

The Verilog code shown below implements an arithmetic function on three 32-bit inputs. What is this function?

```verilog
module mystery_module ( a, b, c, y );
    input [31:0] a, b;
    output [31:0] y;

    wire w;

    assign w = (a > b) ? a : b;
    assign y = (c > w) ? c : w;
endmodule
```

Answer: returns the maximum of \( a \), \( b \), and \( c \)
6. Multiplication Hardware 15 Points

The two unsigned binary numbers shown below are to be multiplied using a standard “shift and add” sequential multiplier and a multiplier that uses Booth’s Algorithm:

```
10011010
X  01111010
```

Answer the following questions:

(a) How many bits will be needed to store the product of these two numbers?

16 bits (15 if you assume the MSB of the multiplier is always 0).

(b) How many additions will be performed by the standard shift-and-add multiplier that multiplies these numbers?

A total of 8 additions are made by conventional multiplication, but shift-and-add only adds when multiplier bits are 1, so this means a total of 5.

(c) How many additions will be performed by the Booth’s Algorithm multiplier that multiplies these numbers?

An addition will be performed at the end of each "run of 1s". Since there are two runs of 1s, there are two additions.

(d) How many subtractions will be performed by the Booth’s Algorithm multiplier that multiplies these numbers?

An subtraction will be performed at the beginning of each "run of 1s". Since there are two runs of 1s, there are two additions.

(e) How many Half Adders and Full Adders would be needed to implement a combinational multiplier that can multiply these numbers?

See the diagram in Lecture 8, page 21. For an N X N bit multiplication, this design requires N-1 adders. The first adder uses two half adders and N-2 full adders. Each following adder uses one half adder and N-1 full adders. So, the total is:

Half-adders: \(2 + (6*1) = 8\)

Full adders: \(6 + (6*7) = 48\)