Review for Exam 2

1. Carry-Look-Ahead addition (CLA)

Understand the formulas for the carry into each one-bit unit from the carry-look-ahead unit for four unit blocks:

\[
\begin{align*}
    c_1 &= g_0 + p_0 c_0 \\
    c_2 &= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
    c_2 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
    c_3 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0
\end{align*}
\]

for the second level

\[
\begin{align*}
    P_0 &= p_3 p_2 p_1 p_0 c_0 \\
    C_0 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0
\end{align*}
\]

And similar for subsequent blocks.

In summary, carry-look-ahead accelerates the calculation of carry bits by using additional logic based on the generation and propagation of carries through blocks of four or eight single bit add units. This can be extended to a similar logic for carries between blocks.

CLA is important to improve the performance (speed) of the ALU for addition and subtraction.

2. Finite state machines (FSM)

Given a description of a situation that can be modeled by an FSM, be able to create a state diagram for it. From the state diagram, be able to build the truth table for state transitions, and from this the PLA to implement the FSM.

Example: Consider a branch prediction mechanism where each branch is recorded in a table including the instruction address, the branch address, and an indication of whether the branch was taken or not the previous two times:

- 0- not taken twice in a row
- 1– taken last time, but not taken the time before
- 2– not taken last time, but taken the time before
- 3– taken twice in a row

If in state 0 or 1 then predict the branch is not taken (access the next instructions in sequence)
If in state 2 or 3 then predict the branch will be taken (access the instructions from the branch address).

Draw this as an FSM where the input to determine next state is whether the branch is taken or not this time. Construct the truth table for state transitions. Show a PLA implementation.
2 bit branch prediction

State for each branch:

00- not taken twice in a row
01– taken last time, but not taken the time before
10– not taken last time, but taken the time before
11– taken twice in a row

FSM partial diagram

\[
\begin{array}{c|c|c}
00 & \rightarrow & 01 \\
\uparrow & \text{take} & \\
\text{not} & \text{not} & \text{taken} \\
& & \downarrow v \\
10 & \leftarrow & 11 \\
\text{not} & & \\
\end{array}
\]

Store in a table

<table>
<thead>
<tr>
<th>address of instruction</th>
<th>branch address</th>
<th>state</th>
</tr>
</thead>
</table>

Each instruction address is checked against first column in table. If a match, we know this is a branch we have seen before, so we use the state to predict:

00 or 01 predict not taken
11 or 10 predict taken

If we predict not taken, continue to feed instruction addresses in order.

If we predict taken, put the branch address as the next instruction address after the branch instruction, and feed that sequence of addresses.

If we mispredict, the instructions following the branch in the pipeline will be wrong and need to be flushed, and the correct sequence of addresses used (This is only the one instruction in our 5-stage pipeline, but could be more in longer pipelines)

Each time a branch is executed, modify the state in the table according to the FSM. If it is not in the table, add it.

Consider other examples of FSMs such as the traffic light example from class or the vending machine example.

Understand how the simple add-shift multiplication circuit works

Understand how we could make multiplication faster by doing adds in parallel. For example for 16 bit multiply would look like this:

Each of the top row of adders would either get the multiplicand or 0 for each input according to the appropriate bit in the multiplier controlling a multiplexor. Then the partial sums would go to the next level, with adjustment for the bits to be in the correct position, and so on through the four levels. Thus 16 bit multiplication takes the time of four additions instead of 16 additions, as in the add-shift multiplication.

How would the same scheme apply to 64 bit multiplication?

How could pipelining be applied when many many multiplications are executed in sequence?
4. Floating point.

For IEEE standard 784 for 32 bit floating point:

High order bit (left-most): 0 positive, 1 negative

Next eight bits are exponent using bias 127 notation (find unsigned decimal value of 8 bits, subtract 127 to get exponent.

Remaining 23 bits are fraction.

Value = sign  1.fraction x 2^exponent

Example 1: find the decimal value represented in floating point by 0x3B680000 (answer may be a value times a power of 2)

00111011011010000000000000000000

0 01110110 110100000000000000000000

Sign is +

Exponent is 64+32+16+4+2 – 127 = 118-127 = -9

Fraction = 1101

1.1101 x 2^-9 = 11101 x 2^-13 = 29 x 2^-13

Example 2: Express -19.75 as a 32 bit floating point value (in hex)

Sign bit is 1.

19 = 16 + 2 + 1  in binary 10011

.75 is .5 + .25 or 0.11 in binary

Overall we get 10011.11 in binary.
Normalized we get   1.001111 x 2^4
Fraction is 00111100…0
Exponent  using 127-bias notation is 4+127 = 131, in binary 10000011

Binary float is 1 10000011 00111100..0
1100 0001 1001 1110 0000 0000 0000 0000

In hex 0xC1E0000
The above diagram shows the circuitry for a single cycle CPU. It could be the basis for several questions. Here are examples:

What is the purpose of each of the red boxes A B C D?

Draw the sub-circuit in the green box indicating what the inputs are and the purpose of the output.

How are multiplexors and decoders used in the blue box, the register file?

What is the purpose of the sub-circuit in the purple box?

This single-cycle CPU provided the basis for the pipeline CPU.

What is added to create a pipeline CPU?

How does the pipeline CPU make execution of a program faster?
For the first simple version of the pipeline CPU we encountered the issue of a branch (control) hazard and the issue of a data hazard.

1. Describe what branch hazard means.
2. Describe what data hazard means.
3. How did we reduce the penalty for a branch hazard?
4. How did we resolve most instances of data hazard?
5. What hazards were left as unavoidable for our 5-stage pipeline?

For any CPU, there can be events that cause computation of the current process to stop. These are called exceptions (or sometimes interrupts).
What are some causes of exceptions?
What happens when an exception occurs?
Why do we need two levels of “privilege” for executing processes? What do these levels designate?