Problem 1: The DLX implementation below lacks bypass paths and, worse than that, lacks the hardware needed for control-transfer instructions.

Changes and cycles shown in blue in the diagram below.

✓ [5 pts] Add exactly the hardware needed so that the control-transfer instructions execute as shown below. Include a connection to the \( =0 \) box used in determining whether a branch is taken.

✓ [5 pts] Add exactly those bypass paths necessary so that the code below executes as shown. Check the code carefully for dependencies, including all those related to the \texttt{jalr} instruction.

✓ [5 pts] Show the cycles in which each added wire will be used.

```plaintext
! Cycle          0  1  2  3  4  5  6  7  8  9 10 11 12
ori r1, r1, #15  IF ID EX ME WB
bnez r1, SKIP    IF ID EX ME WB
add r0, r0, r0   IF IDx
xor r0, r0, r0   IFx
SKIP:
sub r20, r20, r21 IF ID EX ME WB
jalr r20          IF ID EX ME WB
xor r0, r0, r0   IFx
...               
add r15, r31, r0  IF ID EX ME WB
or r16, r16, r15  IF ID EX ME WB
! Cycle          0  1  2  3  4  5  6  7  8  9 10 11 12
```
Problem 2: The DLX implementation below includes bypass paths into the EX/MEM.B register.

[5 pts] Write a program that uses all three paths into the EX/MEM.B register. *Hint: It’s easy.*

```
addi r2, r2, #1 IF ID EX ME WB
sw 0(r1), r2 IF ID EX ME WB
sw 4(r1), r2 IF ID EX ME WB
sw 8(r1), r2 IF ID EX ME WB
```

[10 pts] Design the control logic for the multiplexor feeding the EX/MEM.B register. The control logic should be in the ID stage and feed into the ID/EX.Bx pipeline latch provided in the diagram above.

Changes shown in blue.
Problem 3: Registers $r1$ and $r2$ each contain a signed integer, call them $i$ and $j$. Register $r10$ contains a double-word-aligned address, call the address $A$. Let $p = i * j$.


*Hint: A reasonable solution would use movitof and cvtXtoY instructions.*

! Initially: $r1$ and $r2$ each contain an integer, $i$ and $j$.
! $r10$ contains an address.

! At Mem[$r10$] write $i * j$ (integer format).
! At Mem[$r10+?$] write $i * j$ (double-precision FP)
! At Mem[$r10+??$] write $i * j$ (single-precision FP)

# DLX Solution
movitofp f0, r1
movitofp f2, f2
cvtitod f0, f0
cvtitod f2, f2
multd f4, f0, f2
cvtdtoi f6, f4
sf 0(r10), f6
sd 8(r10), f4
cvtdtos f4, f4
sf 16(r10), f4

# MIPS Solution
mtc1 $1, $f1 # Move to FP (co-processor 1) register.
mtc1 $2, $f2
cvt.d.w $f10, $f1 # Convert from integer to double FP.
cvt.d.w $f20, $f2
mul.d $f4, $f10, $f20
sdc1 $f4, 8($10) # Store double-sized item (from $f4 and $f5).
cvt.s.d $f6, $f4
swc1 $f6, 16($10) # Store word-sized item
cvt.w.d $f6, $f4
swc1 $f6, 0($10)
Problem 4: Answer each question below.

(a) In the two pipelined functional units below an instruction must pass through each segment twice. The A’s are for FP addition and M’s are for FP multiplication.

[Diagram of pipeline units]

☑ [10 pts] Complete the pipeline execution diagram below for a system using these functional units. Don’t overlook the dependency through f6.

! Solution

\[
\begin{align*}
\text{add}d & \ f0, f2, f4 & \text{IF} & \text{ID} & A1 & A1 & A2 & A2 & \text{WB} \\
\text{add}d & \ f6, f8, f10 & \text{IF} & \text{ID} -> & A1 & A1 & A2 & A2 & \text{WB} \\
\text{mul}d & \ f12, f6, f14 & \text{IF} & \text{ID} -> & \text{ID} -> & M1 & M2 & M1 & M2 & \text{WB} \\
\text{mul}d & \ f16, f18, f20 & \text{IF} & \text{ID} -> & \text{ID} -> & M1 & M2 & M1 & M2 & \text{WB}
\end{align*}
\]
(b) The code below, which of course is not DLX, uses memory-indirect and autoincrement addressing.

```
lw r1, @(r2)  ! Memory-indirect load.
lw r4, (r5)+  ! Autoincrement
lh r6, (r7)+  ! Autoincrement
```

[10 pts] Rewrite the code in DLX.

```
lw r1, 0(r2)
lw r1, 0(r1)
lw r4, 0(r5)
addi r5, r5, #4
lh r6, 0(r7)
addi r7, r7, #2
```

(c) The three types of interrupts discussed in class are traps, hardware interrupts, and exceptions.

[6 pts] For each one explain how the exception code (number) is determined.

Traps: the code is specified in the trap instruction itself. Hardware Interrupts: the code is based on the interrupt request line that was asserted. Exceptions: the code is based on what went wrong with the faulting instruction.

[6 pts] For each one explain where control returns after the handler completes.

Trap: The instruction following the trap instruction. Hardware Interrupt: The instruction just after the last one to complete. Exception: the faulting instruction.
(d) An ISA has two implementations, $A$ and $B$; each implementation has a well-written compiler.

\checkmark [5 pts] Would the code compiled by $A$'s compiler run on implementation $B$? Briefly explain.
Yes, since they are compiled for the same ISA.

\checkmark [5 pts] A program is compiled using $A$'s compiler and $B$'s compiler. How might the compiled code differ? Provide a reason for the difference.
Scheduling of instructions might be different because of differences in functional unit latencies. For example, in $A$ loads might have a latency of 1 (as in Chapter 3 DLX) while in $B$ they might have a latency of 2, and so for $B$ the compiler try to move two instructions, rather than one, between a load and a use of the loaded value.

(e) You have become the owner of a large American computer company, congratulations.

\checkmark [6 pts] How can you (legally) influence the decision-making process so that SPEC's next benchmark suite does not unfairly put your company's products at a disadvantage? (Note: Bribery is illegal in the U.S.)
Become a member of SPEC and help create, and vote for, benchmarks fair benchmark suites.
(f) DLX does not have delayed branches, but many other RISC ISAs do.

[6 pts] What is a delayed branch and how does it help?

A delayed branch is one in which the instruction following the branch (or the \(d\) instructions following the branch, though \(d\) is almost always 1) in program order is executed regardless of whether the branch is taken, followed by the branch target if the branch is taken. Many implementations will be forced to squash the instruction following a taken non-delayed branch, with a delayed branch the instruction following the branch is not squashed and so can do useful work.

(g)

[6 pts] How and why is the CPI affected in an implementation re-designed for a higher clock frequency?

With a higher clock frequency less work can be done per clock cycle. This might increase the latency of certain instructions (for example, from six to twelve multiply segments), increasing the number of stall cycles and thus increasing CPI.