Problem M2.1: Execute Data Instructions (Spring 2014 Quiz 1, Part A)

One day, Ben Bitdiddle started an EDSACjr-based company. Ben wanted to leverage the speed of read-only memory and avoid the inherent hazards of the Princeton architecture, so he went with a Harvard architecture. Unfortunately, Ben's system didn't have any index registers, so he couldn't write self-modifying code. That meant there were a large number of programs he couldn't implement anymore. Ben decided to add an instruction to solve this problem. He called his new instruction `EXD` , for execute data. The `EXD` instruction treats the contents of the accumulator as a new instruction and executes whatever that instruction may be. If the accumulator does not contain a valid instruction, then `EXD` falls back on the processor's fault handling for bad instructions (which you needn't worry about).

For example, from Handout #1 the instruction `ADD 6` (which adds the contents of memory at address six to the accumulator) is encoded as: 0000 0100 0000 0110.

Therefore if the contents of the accumulator are 0000 0100 0000 0110, the `EXD` instruction will interpret the accumulator as an `ADD 6` instruction, and add the contents of memory at address six to the accumulator (now interpreted as the *number:* 0000 0100 0000 0110 = 1030). So if memory at address six holds the value one, then the accumulator will become 0000 0100 0000 0111. (Which can be interpreted either as the instruction `ADD 7` or the number 1031.)

To simplify writing assembly code, Ben Bitdiddle also augments the EDSACjr's instruction set with a load instruction, `LD n`. This load simply places the value in memory address n into the accumulator: `ACC` ← `Mem[n]`. LD is encoded as 01011 *n*; that is, the opcode is 01011.

Problem M2.1.A

When Ben shows his idea to Alyssa P. Hacker, she points out that `EXD` could cause an infinite loop. Provide a specific code sequence that illustrates Alyssa's point, using `EXD` to loop forever.

Problem M2.1.B

Ignoring Alyssa's observation, Ben decided to implement his `EXD` instruction for the EDSACjr, but he started having trouble figuring out how to use it. Help Ben by writing a series of EDSACjr instructions that will perform an indirect reduced add (that is, the instructions will take a vector of pointers, follow each pointer, and sum up the values stored at the locations in memory that the pointers specify). In C++, this might look something like:

```
int s=0;
for (int i=0; i < 10; i++){
    s += *A[i];
}
```

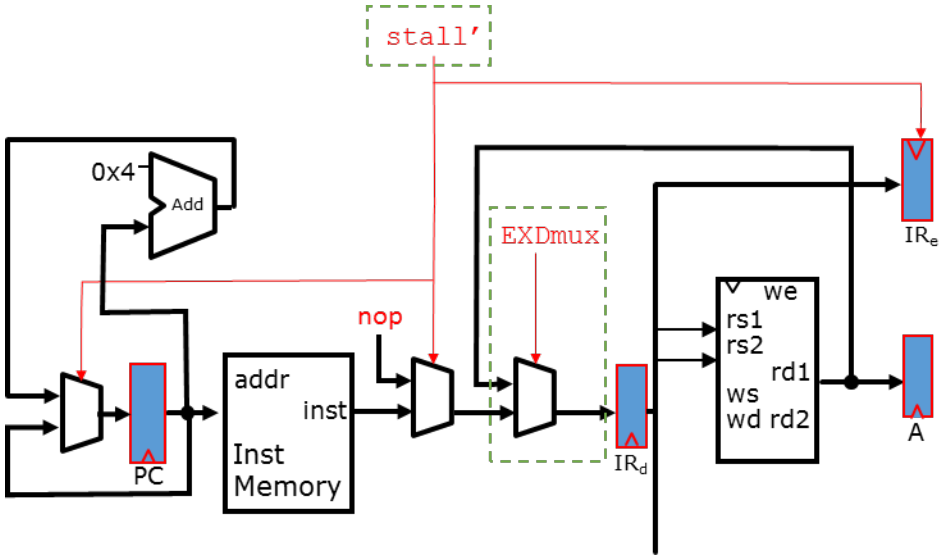
Fill in the template below with assembly code for this program on the Harvard EDSACjr. You can define memory contents for both the data and instruction memories.

```
Data Mem
Addr  Data
A:    120
      107
      122
      130
      151
      112
      132
      109
      140
      117

s:    0
i:    10

107:  40
109:  10
112:  24
117:  50
120:  5
122:  10
130:  20
132:  29
140:  22
151:  12

one:  1
```

```
Instr Mem
Addr  Data
Loop: LD i
      SUB one
      BLT Done
      STORE i




      CLEAR
      BGE Loop
Done: HLT
```

Problem M2.1.C

Unhappy with the performance, now Ben decided to implement a pipelined version of `EXD`. Ben realized that the `EXD` instruction was in and of itself a control hazard. Help Ben safeguard his pipeline. The diagram below shows the front end of the five-stage pipeline we used in class. A new datapath and mux have been added to move `rd1` into the instruction register of the decode stage.



Your task is to write the new stall signal (`stall'`) and fill in the missing signal, `EXDmux`. Write your signal in terms of signals (e.g., `PC` or `rd1` or IR_D) and feel free to use the old stall signal (`stall`).

`stall'` = _____

`EXDmux` = _____

Problem M2.2: CISC and RISC: Comparing ISAs

This problem requires the knowledge of Handout #2 (CISC ISA—x86jr), Handout #3 (RISC ISA—MIPS32), and Lectures 1 and 2. Please read these materials before answering the following questions.

Problem M2.2.A — CISC

Let us begin by considering the following C code.

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i = 0; i<b; i++){
    result=result+a;
  }
}
```

Using gcc and objdump on a Pentium III, we see that the above loop compiles to the following x86 instruction sequence. (On entry to this code, register `%ecx` contains `i`, register `%edx` contains `result` and register `%eax` contains `a`. `b` is stored in memory at location `0x08047580`.) A brief explanation of each instruction in the code is given in Handout #2.

```
            xor    %edx,%edx
            xor    %ecx,%ecx
 loop:      cmp    0x08047580,%ecx
            jl     L1
            jmp    done
 L1:        add    %edx,%eax
            inc    %ecx
            jmp    loop
 done:    ...
```

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Problem M2.2.B — RISC

Translate each of the x86 instructions in the following table into one or more MIPS32 instructions in Handout #3. Place the L1 and loop labels where appropriate. You should use the minimum number of instructions needed. Assume that upon entry R2 contains `a` and R3 contains `i`. R1 should

be loaded with the value of b from memory location 0x08047580, while R4 should receive `result`. If needed, use R5 to hold the condition value and R6, R7, etc., for temporaries. You should not need to use any floating point registers or instructions in your code.

x86 instruction	**label**	**MIPS32 instruction sequence**
`xor %edx,%edx`		
`xor %ecx,%ecx`		
`cmp 0x08049580,%ecx`		
`jl L1`		
`jmp done`		
`add %eax,%edx`		
`inc %ecx`		
`jmp loop`		
`...`	*done:*	`...`

How many bytes is the MIPS32 program using your direct translation? How many bytes of MIPS32 instructions need to be fetched for b = 10 using your direct translation? How many bytes of data memory need to be fetched? Stored?

Problem M2.2.C — Optimization

To get more practice with MIPS32, optimize the code from part **B** so that it can be expressed in fewer instructions. Your solution should contain commented assembly code, a paragraph which explains your optimizations and a short analysis of the savings you obtained.

Problem M2.3: Addressing Modes on MIPS ISA

Ben Bitdiddle is suspicious of the benefits of complex addressing modes. So he has decided to investigate it by incrementally removing the addressing modes from our MIPS ISA. Then he will write programs on the "crippled" MIPS ISAs to see what the programming on these ISAs is like.

Problem M2.3.A — Displacement addressing mode

As a first step, Ben has discontinued supporting the displacement (base+offset) addressing mode, that is, our MIPS ISA only supports register indirect addressing (without the offset).

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why.

`LW R1, 16(R2)` ➔

Problem M2.3.B — Register indirect addressing

Now he wants to take a bolder step by completely eliminating the register indirect addressing. The new load and store instructions will have the following format.

```
LW R1, imm16    ; R1 <- M[imm16]
SW R1, imm16    ; M[imm16] <- R1
```

6	5	5	16
Opcode	**Rs**		**Offset**

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why. (Don't worry about branches and jumps for this question.)

`LW R1, 16(R2)` ➔

Problem M2.3.C **Subroutine**

Ben is wondering whether we can implement a subroutine using only absolute addressing. He changes the original ISA such that all the branches and jumps take a 16-bit absolute address (the 2 lower orders bits are 0 for word accesses), and that `jr` and `jalr` are not supported any longer.

With the new ISA he decides to rewrite a piece of subroutine code from his old project. Here is the original C code he has written.

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i=0; i<b; i++){
    result=result+a;
  }
}
```

The C code above is translated into the following instruction sequence on our original MIPS ISA. Assume that upon entry, R1 and R2 contain `b` and `a`, respectively. R3 is used for `i` and R4 for `result`. By a calling convention, the 16-bit word-aligned return address is passed in R31.

```
Subroutine: xor  R4, R4, R4   ; result = 0
            xor  R3, R3, R3   ; i = 0
loop:       slt  R5, R3, R1
            bnez R5, L1       ; if (i < b) goto L1
return:     jr   R31          ; return to the caller
L1:         add  R4, R4, R2   ; result += a
            addi R3, R3, #1   ; i++
            j    loop
```

If you can, please rewrite the assembly code so that the subroutine returns without using a `jr` instruction (which is a register indirect jump). If you cannot, explain why.

Problem M2.4: Write Effective Address Extensions (Spring 2014 Quiz 1, Part B)

You've noticed that many programs execute code similar to the following during loops:

```
LD  R1, 4(R2)
ADD R2, R2, 4
```

Or:

```
ST  R1, 4(R2)
ADD R2, R2, 4
```

You want to optimize your architecture for this common case. You are going to do so by adding "write effective address" variants of the load and store instructions, `LDWA` and `STWA`. The semantics of these instructions are that they will perform the normal memory operation (`LD` or `ST`) and then write the effective address in the register that indexed into memory (*not* the register whose contents are read/written to memory). Specifically these instructions do the following:

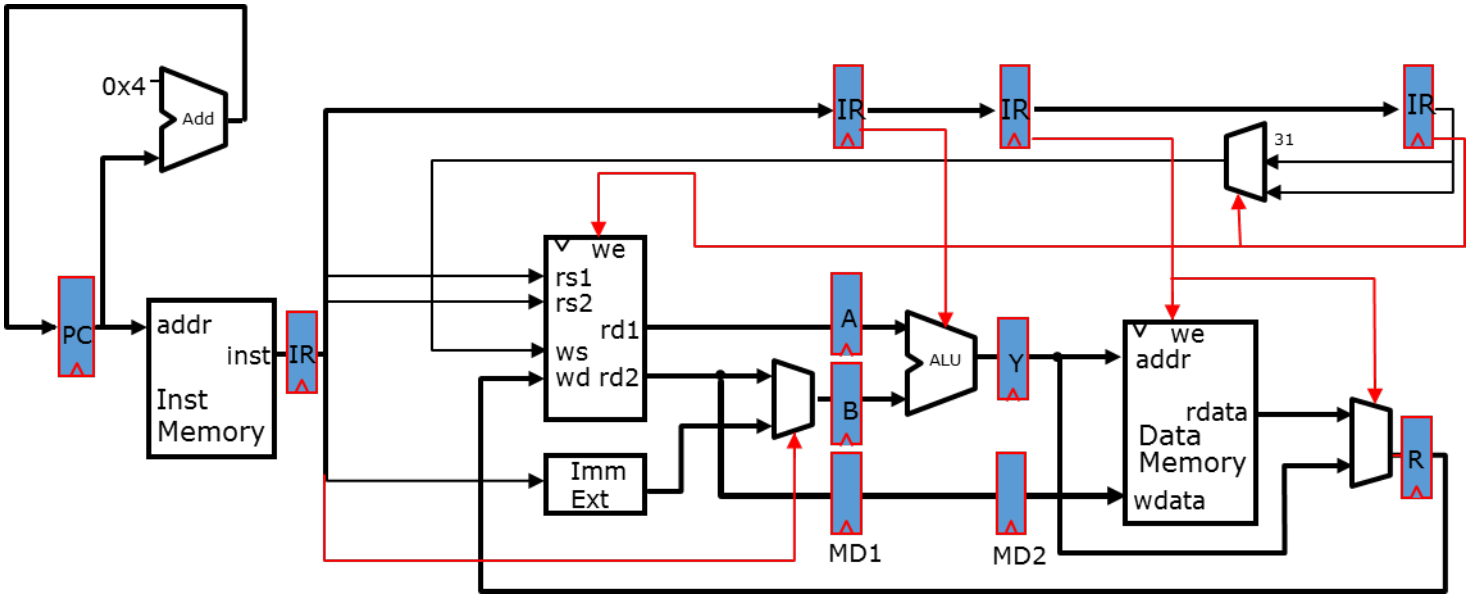
```
LDWA rs, rt, Imm:
     rs ← Memory[(rt) + Imm]
     rt ← (rt) + Imm

STWA rs, rt, Imm:
     Memory[(rt) + Imm] ← (rs)
     rt ← (rt) + Imm
```

These extensions allow us to rewrite the previous examples as:

```
LDWA R1, R2, 4
```

And:

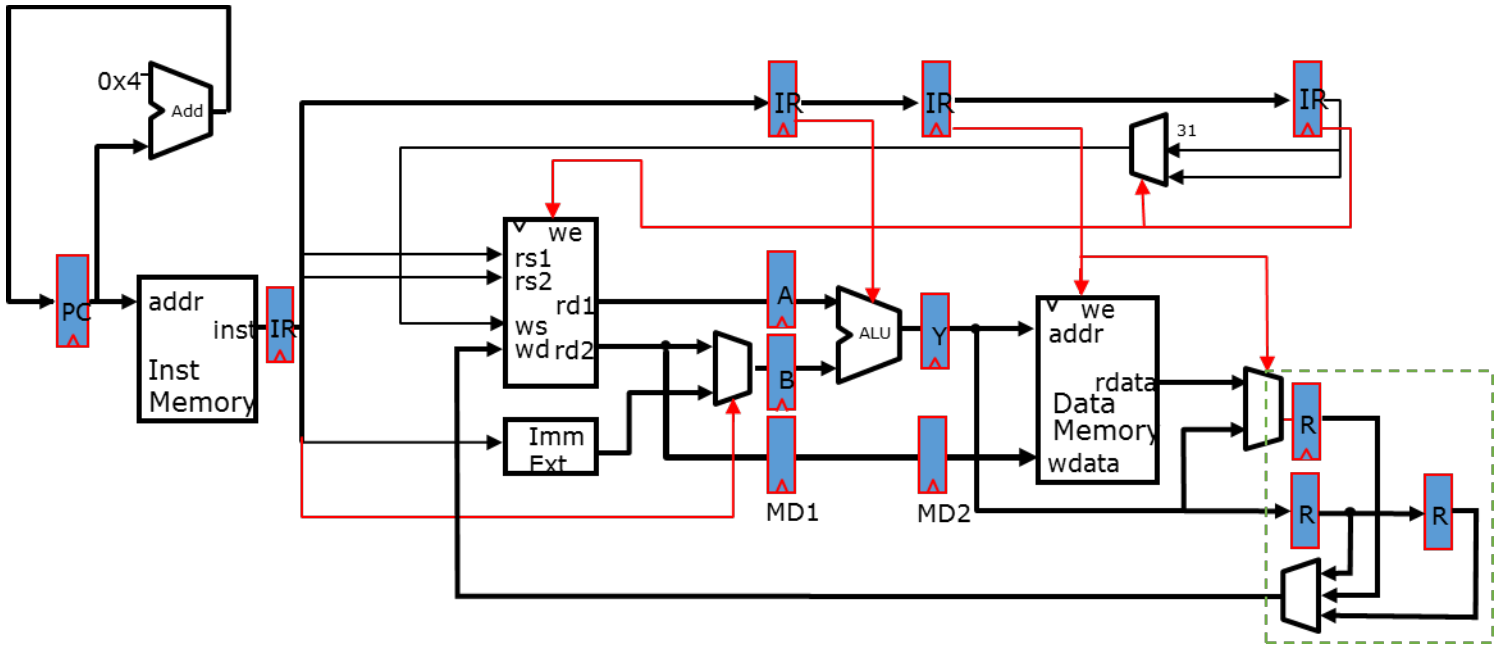
```
STWA R1, R2, 4
```



Problem M2.4.A

You start with implementing `STWA`. For the following sequence of instructions and the standard five-stage pipeline (shown above), indicate how each instruction will flow through the pipeline on the following page. Assume full bypassing and stall logic are implemented for your architecture. Use arrows to indicate forwarding and dashes for stalls, as illustrated.

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
`LD R1, 0(R2)`	F	D	E	M	W								
`ADD R3, R1, 10`		F	D	-	E	M	W						
`LD R4, 0(R3)`													
`STWA, R4, R1, 4`													
`STWA R4, R1, 4`													
`ADD R2, R1, R3`													



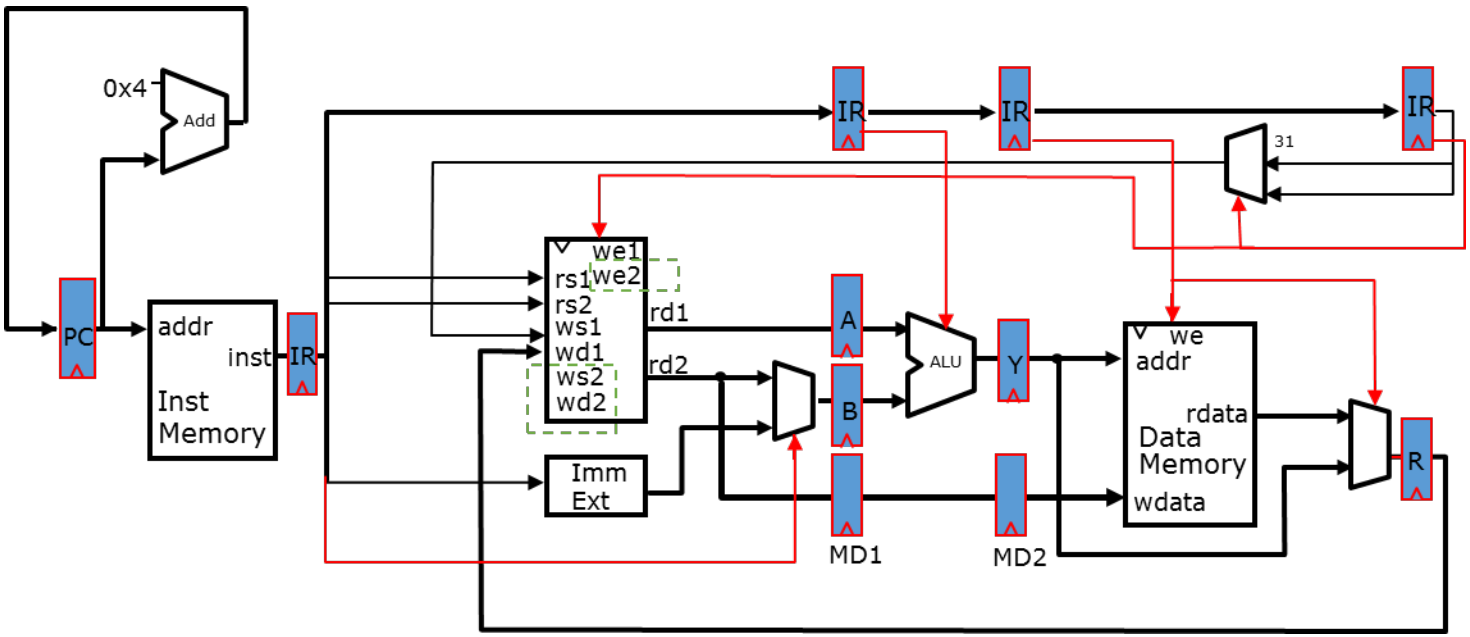
Problem M2.4.B

You next want to implement `LDWA`, and quickly realize that `LDWA` runs into a structural hazard on the register file. You decide to fix this by adding an extra writeback stage (W2) to your pipelined design as shown above. In one or two sentences, explain what the hazard is and why the additional stage fixes it (assume correct stall logic).

Problem M2.4.C

Assume that the six-stage design above has full bypassing and correct stall logic. Fill in the pipeline for the instructions given below, using arrows and dashes as before.

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
`LD R1, 0(R2)`	F	D	E	M	W1	W2							
`ADD R3, R1, 10`		F	D	-	E	M	W1	W2					
`LDWA R5, R3, 0`													
`ADD R1, R3, R4`													
`LDWA R5, R3, 0`													
`ADD R1, R5, R0`													



Problem M2.4.D

Adding a second writeback stage is only one way to fix this structural hazard. An alternative design is to add a second write port to the register file. Quickly sketch the datapath for this design in the diagram above. You do *not* need to write the stall logic. (Additional signals are: `we2`, `ws2`, `wd2`.)

Problem M2.4.E

In one or two sentences, explain the tradeoffs between adding an additional pipeline stage vs. adding a write port to the register file. What conditions might favor one or the other design?